Universidade Estadual de Campinas
Instituto de Computação

Alexandre Melo Braga

Towards the Safe Development of Cryptographic Software

Rumo ao Desenvolvimento Seguro de Software Criptográfico

CAMPINAS
2017

Alexandre Melo Braga

Towards the Safe Development of Cryptographic Software

Rumo ao Desenvolvimento Seguro de Software Criptográfico

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Ricardo Dahab

Este exemplar corresponde à versão final da Tese defendida por Alexandre Melo Braga e orientada pelo Prof. Dr. Ricardo Dahab.

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): CNPq, 312642/2015-6

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Silvania Renata de Jesus Ribeiro - CRB 8/6592

B73t

Braga, Alexandre Melo, 1974-
Towards the safe development of cryptographic software / Alexandre Melo Braga. – Campinas, SP : [s.n.], 2017.

Orientador: Ricardo Dahab.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia. 2. Engenharia de software. 3. Tecnologia da informação - Segurança. 4. Software - Desenvolvimento - Medidas de segurança. 5. Engenharia de software - Medidas de segurança. I. Dahab, Ricardo, 1957-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Rumo ao desenvolvimento seguro de software criptográfico
Palavras-chave em inglês:
Cryptography
Software Engineering
Information technology - Security
Software - Development - Security controls
Software engineering - Security controls
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Ricardo Dahab [Orientador]
Alessandro Fabricio Garcia
Ricardo Felipe Custódio
Breno Bernard Nicolau de França
Diego de Freitas Aranha
Data de defesa: 19-12-2017
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Alexandre Melo Braga

Towards the Safe Development of Cryptographic Software

Rumo ao Desenvolvimento Seguro de Software Criptográfico

Banca Examinadora:

- Prof. Dr. Ricardo Dahab
  Instituto de Computação, Universidade Estatual de Campinas
- Prof. Dr. Alessandro Fabricio Garcia
  Pontifícia Universidade Católica do Rio de Janeiro
- Prof. Dr. Ricardo Felipe Custódio
  Laboratório de Segurança Em Computação, Universidade Federal de Santa Catarina
- Prof. Dr. Breno Bernard Nicolau de França
  Instituto de Computação, Universidade Estatual de Campinas
- Prof. Dr. Diego de Freitas Aranha
  Instituto de Computação, Universidade Estatual de Campinas

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 19 de dezembro de 2017

Dedication

To Kelcy, Gustavo, and Leonardo.

Follow your bliss
and the universe will
open doors where there
were only walls. (Joseph Campbell)

Acknowledgements

To my wife.
To my parents.
To my family.
To my advisor and my friend.
To my fellows from Coimbra.
To my defense's committee.
To my interns.
To my bosses.
To UNICAMP.
To CPqD.

I've never been alone on this journey.
Like an apprentice with many masters.

My achievement had the support of each and every one of you.
Like a pilgrim with many helpers.

Thanks for being part of my adventure.
Like a hero with many faces.

Resumo

A sabedoria popular diz que qualquer sistema é tão seguro quanto o seu elo mais fraco. Com o uso generalizado de bibliotecas criptográficas de boa reputação, baseadas em padrões confiáveis de algoritmos de qualidade, implementações inseguras ou falhas na matemática subjacente não são os elos fracos mais prováveis. Porém, o comprometimento da criptografia ou da sua implementação são eventos raros de impacto catastrófico.

Atualmente, a investigação sistemática de problemas práticos associados ao mau uso de criptografia está ganhando força. À medida que a segurança torna-se transparente para os usuários e a criptografia de boa reputação está disponível para todos os desenvolvedores, o elo fraco mais provável deixa de ser a infra-estrutura criptográfica para se tornar o software em torno da criptografia, escrito por desenvolvedores não especialistas no assunto. Hoje, o mau uso generalizado da criptografia em software é a fonte mais freqüente de problemas de segurança relacionados à criptografia. O efeito cumulativo desse mau uso generalizado também tem um impacto catastrófico, embora difuso.

Esta tese investiga o papel da criptografia na segurança de software e fortalece o surgimento da **segurança de software criptográfico** como um novo campo de estudo preocupado com o desenvolvimento sistemático de software criptográfico seguro.

Esta tese alcançou os seguintes resultados. Primeiramente, uma revisão da literatura sobre programação e verificação de software criptográfico descobriu que apenas um quarto das ferramentas identificadas poderia ser usado por programadores não especialistas em criptografia. Em segundo lugar, uma metodologia para o desenvolvimento de software criptográfico seguro consolidou as práticas casuais empregadas pela segurança de software na construção de software criptográfico. Adicionalmente, um estudo empírico sobre comunidades on-line para programação descobriu que o mau uso de criptografia é muito freqüente, com padrões recorrentes de mau uso. Em quarto lugar, uma avaliação de ferramentas de análise estática descobriu que estas ferramentas detectam pouco mais de um terço dos maus usos criptográficos. Além deste, um estudo longitudinal e retrospectivo sobre o mau uso da criptografia descobriu que desenvolvedores podem aprender a usar APIs criptográficas sem realmente aprender criptografia, enquanto alguns maus usos persistem ao longo do tempo. Finalmente, uma classificação dos maus usos da criptografia, voltada para a área de segurança de software, e uma metodologia para o desenvolvimento de software criptográfico compõem o corpo de conhecimento para o seu desenvolvimento seguro.

Há uma grande lacuna entre o que os especialistas em criptografia vêem como maus usos de criptografia, os maus usos que as ferramentas de segurança atuais são capazes de detectar e aquilo que os desenvolvedores vêem como uso inseguro da criptografia. O arcabouço conceitual proposto nesta tese contribui para preencher essa lacuna, aplicando a segurança de software criptográfico no desenvolvimento de software seguro.

Abstract

Conventional wisdom says that any system is only as secure as its weakest link. With the widespread use of good-standing cryptographic libraries, based upon standards of scrutinized algorithms, insecure implementations or flaws in the underlying mathematics are not the likely weakest links. The compromise of cryptography or of its implementations are rare events with catastrophic impacts, though.

Nowadays, the systematic investigation of real-world issues associated to cryptography misuse is gaining momentum. As security becomes transparent to end-users and good-standing cryptography is readily available to every developer, the likely weakest link moves from cryptographic infrastructure to the code surrounding cryptography, written by ordinary developers, non-experts in cryptography. Today, the widespread misuse of cryptography in software is the most frequent source of security issues related to cryptography. The cumulative effect of this widespread misuse also has a catastrophic impact, although a diffused one.

This thesis investigates the role of cryptography in the discipline of software security and strengthens the emergence of **cryptographic software security** as a new field concerned with the development of secure cryptographic software.

This thesis achieved the following results. First, a literature review about programming and verification of secure cryptographic software found that only one quarter of surveyed tools could be used by non-expert developers working above crypto APIs. Second, a methodology for development of secure cryptographic software captured the casual practices employed by software security when building cryptographic software. Third, an empirical study in online communities for programming found that cryptography misuse is frequent, showing recurring patterns of misuse. Fourth, an experimental evaluation of static analysis tools for cryptography found that evaluated tools detected a little more than one third of crypto misuses. Fifth, a longitudinal and retrospective study on how developers misuse cryptography found that developers can learn how to use crypto APIs without actually learning cryptography, while some misuse persists over time. Sixth, a systematic classification of cryptography misuse for software security and a methodology for developing secure cryptographic software compose a body of knowledge for the field of cryptographic software security.

There is a huge gap among what cryptography experts see as cryptography misuse, what current tools are able to detect, and what developers see as insecure use of cryptography. The conceptual framework in this thesis contributes to bridge this gap by applying cryptographic software security in order to safely use cryptography when developing secure software.

Contents

Chapter 1

Introduction

Security controls based upon cryptography are now common in software systems, available to every developer, and affecting the life of many users of this technology. For instance, today, we can see in every smart phone the end-points of crypto systems transparently managing millions of keys on behalf of their users, in publicly available applications [107, 95, 102], going beyond those traditional use cases for cryptography, and sometimes, with social and political implications [99].

Evidence for the widespread use of cryptography can be easily found in popular mobile apps allegedly using cryptography to secure communication [107, 95, 102], in platforms for mobile devices that offer full-disk encryption [98], and in simple searches for keywords like *cryptography* and *encryption* on well-known app stores, which show hundreds of results for mobile apps related to cryptography [23].

Cryptography is an essential part of software security. However, it is one of the most misunderstood technologies in the development of secure systems [68, 46, 38, 52, 27]. Its correct use is full of design decisions and coding pitfalls that challenge software developers, security researchers, and tool builders.

For almost thirty years, studies have shown that vulnerabilities in cryptographic software have been mainly caused by software defects and poor management of cryptographic parameters and other sensitive material [12, 90, 89, 91, 56].

Nowadays, the systematic investigation of real-world issues associated to cryptography misuse is gaining momentum, as many cryptography experts, including renowned researchers (for instance, Ross Anderson [13], Dan Boneh [54], Phillip Rogaway [87], Pascal Junod [62], Matthew Green [55], Daniel Bleichenbacher [17], and Simson Garfinkel [6]), believe now that cryptographic software security is not only a matter of secure implementation of cryptographic algorithms encapsulated by rich APIs, but also requires a system's thinking approach that includes the correct usage of these APIs, conscious system design, adequate verification technologies, and appropriate expert help, all aiming at avoiding cryptography misuse.

However, in general, security experts (including cryptography experts, in particular) are rare [51], adequate tools are barely available [25, 31], specialized literature is arcane (from a developers' viewpoint) [26], and online resources usually give misleading advice [27, 29]. Therefore, developers are left unassisted most of the time when building software with cryptographic features, resulting in the widespread misuse of cryptography.

For instance, recent studies found similar numbers for how frequent cryptography misuse is in ordinary software. Lazar et al. [68] found that 83% of cryptography vulnerabilities are due to misuses of crypto libraries, while Egele et al. [46] and Chatzikonstantinou et al. [38], in two different studies, found that about 88% of mobile apps (from a sample of Google's market place) showed some cryptography misuse. Also, Gajrani et al. [52] found that 90% of apps from diverse app stores are exploitable because of cryptographic vulnerabilities. Our own results [27] confirmed these high values for cryptography misuse. We found that cryptography misuse appears in regular discussions about cryptographic programming in online communities in 90% of posts for a Java community and 71% of posts for an Android community.

Cryptography misuse in software has many facets and causes. In general:

- Cryptologists do not know or do not care about programming and software development issues [13, 87, 55, 62].
- Developers of crypto libraries do not design cripto APIs to foster correct use of cryptography [6, 62].
- There is a lack of documentation to foster proper use of cryptography in coding [6].

We augmented this list with our own findings:

- Developers learn how to use crypto APIs without actually learning cryptography correctly [29].
- Methods for software security overlook the development of cryptographic software [28].
- Vulnerability detection tools are better in finding other security issues then in detecting cryptography misuse [31].
- Developers are not exposed to comprehensive examples of cryptography misuse [30].

This introductory chapter proceeds as Section 1.1 explains concepts and terminology adopted in this text and Section 1.2 explains the thesis and its structure.

1.1 Terminology adopted by this text

In multidisciplinary texts, it is common to find distinct understandings to similar concepts adopted by different domains. Particularly, applied cryptography is full of domain-specific jargon and acronyms that are difficult for outsiders to understand. This section does not give an introduction to cryptography and its nomenclature. For that, we redirect the reader to appropriate sources [72, 50, 26]. Instead, this section introduces common concepts and terminology adopted throughout this text and that are discussed in more detail in Chapter 3.

Cryptographic software (crypto software, for short) is software that preserves major security goals (namely, confidentiality, integrity, authenticity, and non repudiation) transparently blended into functionality, by using cryptographic methods available through

reusable libraries and frameworks. In this context, **applied cryptography** is the use of cryptographic infrastructures (e.g., packages, libraries, and APIs) to build cryptography-based security into software.

We distinguish between cryptographic software and **cryptography software**, where the later is related to the traditional concept of software for cryptography that implements only cryptography functions (e.g., encryption, signing, secure hashing, etc.) to be used by other software as external components, services, or infrastructure. Cryptography software usually offers generic functions that contribute to specific features of applications. Examples of cryptography software are cryptographic libraries as well as packages for secure communication with TLS.

Cryptography misuse (crypto misuse, for short) is a programming bad practice frequently found in cryptographic software, leading to vulnerabilities introduced by developers during coding tasks associated to use cases enabled by cryptography. In many cases, crypto misuse is also associated to design flaws and insecure architectural choices. For instance, a hard-coded encryption key found in code means that a key management system is missing.

Crypto misuse is not related to implementation of cryptographic algorithms. Instead, crypto misuses emerge when ordinary developers use cryptographic infrastructures in daily coding activities during the development of cryptographic software. Every crypto misuse has its corresponding good (a.k.a. correct) use of cryptography and *vice-versa*. A crypto misuse can manifest itself in different forms and be instantiated in many ways, depending on technology specifics.

Cryptography-based security (a.k.a. cryptographic security) is security provided by the correct use of cryptography. This is an emergent property of systems and can only be determined in the context of the whole software and its environment, e.g., by how a crypto system interacts with its surrounding software.

Cryptographic programming is the programming of cryptographic features by means of common coding tasks associated to cryptographic use cases. Cryptographic programming can benefit from both secure coding of crypto software and coding secure crypto software. **Secure coding** of crypto software uses general secure programming techniques during the coding of crypto software to avoid common coding bugs (for instance, buffer overflows and dead code) that can compromise crypto software. **Coding secure crypto software** begins where the latter ends and embraces specific secure coding techniques and programming tricks to better defend crypto software against cryptography misuse.

Cryptographic vulnerability is a defect in cryptographic software that an adversary can exploit in actual attacks and is specifically associated to crypto misuse introduced during coding tasks when implementing cryptographic use cases.

Cryptographic feature is a characteristic of a software functionality that results from the successful execution of cryptographic use cases in cooperation with other software functions. It is commonly associated to the security goals of cryptography. For instance, confidentiality of conversation, integrity of data, authenticity of origin, and non repudiation of authorship are all cryptographic features associated to software functionality.

Cryptographic use case is a sequence of actions or steps performed by a crypto-

graphic software to achieve one or mode security goals associated with cryptography (e.g., confidentiality, integrity, authenticity, and non repudiation). The cryptographic use cases commonly found in software are encrypting data at rest, secure communication, password protection, and authentication or validation of data. In **modern cryptographic software**, crypto use cases are transparently blended to the application's functionality, becoming invisible to end-users, as well as implementing unconventional features and uncommon use cases.

Cryptographic (coding) task is an activity that needs to be accomplished when coding cryptographic software, so as to produce the successful implementation of a security goal related to cryptography. Cryptographic coding tasks commonly found in software are the following: encryption and decryption; generation of digital signatures or authentication codes and their verification; key generation or agreement; secure channels based on SSL/TLS; and handling and verification of digital certificates. There are also supporting tasks, roughly associated to key management, such as storage, recovery, distribution, and revocation of keys and certificates.

1.2 Structure of this thesis

This is a thesis by publication (a.k.a. a compilation thesis) in which the body of the text is a verbatim compilation of published contributions or, in case of journal papers, submitted to publication. This section explains the thesis and its structure, according to the following topics: purpose, scope, research methodology, published contributions, main findings, main conclusions, and future improvements.

1.2.1 Purpose and scope

The **purpose** of this thesis is twofold. First, to investigate the role of cryptography in the broad field of software security. Second, to strengthen the emergence of the field of **cryptographic software security**, which is concerned with the development of secure cryptographic software. At first glance, this scope seems too broad. However, there is a great deal of evidence that cryptographic software security is in fact an emergent field looking for systematization. This thesis contributes to a better understanding of how cryptographic software security can be structured as a field of study.

The **scope** of the thesis is the development of cryptographic software from the application perspective (above and beyond crypto APIs) and the viewpoint of the software developer. Central to this scope is the investigation of how cryptography is misused in practice as a phenomenon. Another concern is to understand how security experts can better prevent and detect cryptography misuse in software systems.

In order to understand the developer's point of view, we (cryptography experts) tried to avoid an adversarial attitude against developers (that attitude by which many experts try to build developer-proof cryptography and penalize developers for not knowing the tricky details of applied cryptography) and put ourselves in their shoes.

In this thesis, we explored three research topics:

1. The occurrence of cryptography misuse in online communities for programming.
2. The support of automated tools for detection of cryptography misuse in software.
3. The attention deserved by cryptography in development methods for secure software.

1.2.2 Research methodology and main publications

The **research methodology**, in general, followed three main activities: literature review, technology survey, and the proposal of methods for secure cryptographic software.

The first activity (literature review) consisted in identifying and studying related literature. The activity was planned as follows:

- Study of good uses and misuses of cryptography in software applications.
- Study of tools for secure development of cryptographic software.
- Study of methods for software security applied to cryptographic software.

The second activity (technology survey) consisted in prospecting modern cryptographic software and analyzing their uses of cryptography. This activity was planned as a study of the occurrence of cryptographic bad practices in development forums for modern software platforms.

The third activity (the proposal of methods for secure cryptographic software) focused on a broad methodology for building secure cryptographic software. It was accomplished in three phases, as follows:

- Elaboration of a layered architecture for cryptographic software.
- Elaboration of a framework for quality assurance of cryptographic software.
- Elaboration of a methodology for development of secure cryptographic software.

The research methodology described above reflects the planned division of work, while the actual flow of work suffered adaptations over time, as expected, and is illustrated by the presentation order of published papers. The chronological order of publications reflects the evolution of our understanding on the subject and provides the reader with an opportunity to follow the steps we went through in this research. It is also worth noting that the Ph.D. candidate (the author of this thesis) is the main author of all these papers.

A preliminary publication, not included in this compilation, was a book chapter introducing cryptography to software developers [26], distinguishing good uses and misuses of cryptography with examples in Java and its crypto API.

The **compilation of papers** for this thesis is composed of six published works:

1. A survey of research papers about programming and verification of secure cryptographic software [25].
2. A methodology for development of secure cryptographic software [28], which received the award for best student paper at ICSSA'2016.

3. An empirical study on cryptography misuse in online communities [27].

4. An experimental evaluation of static code analysis tools concerning the support given by these tools to cryptography [31].

5. A longitudinal and retrospective study in online communities on how developers misuse cryptography over time [29].

6. A classification of cryptography misuse for software security and the refinement of the working methodology to better fit cryptography in development of secure software [30].

1.2.3 Findings and contributions

Here we describe the main **findings** and **contributions** of this thesis by publication.

First, when reviewing the research literature about tools for programming and verification of secure cryptographic software [25], we found that only one quarter of all tools could be used above crypto APIs, with the other three quarters consisting of domain-specific tools for secure implementation of cryptographic algorithms, mainly applied below crypto APIs. These tools were all academic prototypes showing preliminary results, suggesting that secure coding for cryptography was an emerging topic, lacking support from commercial tools. Also, we informally observed that none of the tools and techniques could, by themselves, satisfy the whole landscape of development scenarios for cryptographic software.

Second, when studying how traditional methods for software security treated cryptography, we devised a working methodology for Development of Secure Cryptographic Software (DSCS) [28], which provided an ordered way to approach cryptography into Secure Software Development Life Cycles (SSDLC). Our methodology captured the casual practices of secure software development employed by developers when building cryptographic software. This methodology was based upon our previous experience in building cryptographic software, generalizing several practices we have seen in actual software developments, as well as practical evidence of cryptography misuse.

Third, when investigating online communities to learn how developers misuse cryptography [27], we found that several types of cryptography misuse frequently appeared in the communication among developers with high probabilities (90% for Java and 71% for Android). By associating cryptography misuse to platform-specific issues, in three different programming communities, we were able to identify recurring patterns of cryptography misuse: specific misuses associated to each other in pairs or triples, including worst-case scenarios, when at least three misuses appeared together, in the same piece of code, with relatively high probability, and related to specific use cases or coding tasks.

In the process of investigating cryptography misuse in online communities, we devised a method to extract association rules among types of cryptographic misuse. This method applied a data mining technique named *Apriori* and, as far as we know, was the first attempt to customize data mining techniques to learn association rules from developer's misbehavior when coding cryptographic software.

Fourth, when benchmarking static code analysis tools for cryptography [31], we performed the first practical evaluation of static code analysis tools concerning the support given by these tools to cryptography usage, including a detailed set of test cases for cryptography misuse [22, 21]. In this experiment, we evaluated five free tools in order to answer the questions of how and to which extent cryptography misuse was caught by free tools currently available to developers.

In this benchmarking, we found that the union of misuses detected by all five tools covered only about 35% of crypto misuses in our test cases. We also found that, in general, evaluated tools performed better in simple misuses regarding weak cryptography and bad randomness, and worse in issues for key management and program design flaws. Additionally, we found that the evaluated tools were unable to detect non-trivial misuses for insecure curve selection in Elliptic Curve Cryptography (ECC), weak parameters for key agreement with Diffie–Hellman (DH) and ECDH, misconfigured digital signatures with ECDSA, and many insecure configurations for RSA.

Cryptographic software security seems to always require expert help to some extent in order to assure quality in different moments of development. Understanding that cryptography experts are rare and that automated tools do offer useful support to both developers and security experts, despite the huge gap between cryptography knowledge and what tools currently detect, we also proposed a benchmarking methodology intended to find the adequate tool for specific development contexts, considering three aspects: team's skills in cryptography, (un)availability of experts, and the complexity of the target application. Our methodology was able to find the best suited (free) tools for these distinct usage scenarios.

Fifth, when continuing our investigation about cryptography misuse in online communities [29], we performed a longitudinal and retrospective study by tracking users' activities over time, from the data provided by our third contribution [27]. We found that the use of weak cryptography (e.g., broken algorithms or misconfigured implementations of standards) is not only common in online communities, but also recurrent in developers' discussions, suggesting that developers learn how to use crypto APIs without actually learning cryptography. We also found that the lack of knowledge in cryptography is a recurrent source of coding bugs in API usage and does not depend on how long developers use cryptography APIs.

In this study, which we believe was the first of its kind, we concluded that users of online communities were not learning the tricky details of applied cryptography, despite their immediate gains in solving programming issues related to cryptographic APIs. Again, in the process of investigating user's lifespans, we devised a method based upon data mining techniques for clustering developers' posts from their asynchronous lifespans into a comprehensive life cycle. This life cycle was used to compare users who were active in different time periods.

Sixth, when assembling a body of knowledge for cryptographic software security [30], we correlated our previous contributions [25, 27, 29, 31], using two empirical studies [27, 29] and one experimental evaluation [31] to validate our classification of cryptography misuse, and refined our methodology [28] to better fit cryptography into secure software development.

In this investigation [30] we also detailed two mapping studies. The first explored the broad field of software security to understand the established knowledge on how cryptography is approached by software security. The second mapping study complemented the first and explored the cryptography knowledge found in software security textbooks, systematizing our classification of cryptography misuse for software security. In this text, we contribute to strengthen the field of development of secure cryptographic software.

Other two important **contributions** of this thesis are the following. Besides all the above mentioned findings and contributions, this thesis has two byproducts that gradually emerged from this investigation. With time, these two working products became central to our work.

The first is the classification of cryptography misuse for software security that has been assembled and refined from the very beginning. We can map the root ideas of it to previous work [20], when we proposed design patterns to cryptographic software, and preliminary results [26], when we trained developers on how not to use cryptography. Since then, it has influenced and been influenced by findings from investigations of online communities [27, 29] and static analysis tools [31], in successive refinements.

The second is the methodology for development of secure cryptographic software. Over the years, as our understanding evolved, the methodology has steadily matured in such a way that now we can clearly see two versions of it: the first one [28] based upon our previous experience in building cryptographic software (documented by many development cases [32, 34, 37, 24, 35]) and empirical studies [27, 29]; and the second [30], also supported by experimental evaluation[31] of tools and systematic mapping studies[30].

1.2.4 Main conclusions

The general and emergent **conclusions**, cross-findings of all contributions of this thesis, are the following.

We concluded that there is a huge gap among what cryptography experts see as cryptography misuse, what tools are able to detect, and what developers see as insecure programming for cryptography. For instance, we found many blind spots in tools' coverage of cryptography misuse, suggesting that tool builders have not payed enough attention to cryptographic software. Interestingly, these blind spots were barely mentioned by developers in online communities.

We also concluded that the existing knowledge gap we identified between what cryptography experts promote as secure cryptography and what tool builders and software developers actually see as cryptographic software security gives enough evidence for the need for a systematic approach to cryptography into software security, a new field of study, which we name **cryptographic software security**. Cryptographic software security is an emergent and multidisciplinary field of study located on the intersection of three domains: software engineering, software security, and applied cryptography.

We understand now that there is more in cryptographic software security than, for instance, coding best practices to avoid obsolete cryptography or misconfigured algorithms. We not only found statistically relevant associations among categories of cryptography misuse, but also identified a hierarchical relationship among groups of misuses, relating

them to code, design, and architecture. With these relations in mind, we were able to better understand the roles of tools and experts in the development of secure cryptographic software.

Also, we found that static analysis tools favor the detection of simple crypto misuse within most frequent use cases and coding tasks, suggesting a prioritization of efforts by tool builders, in order to offer a trade-off between supporting simple misuses in frequent use cases and coding tasks, against neglecting sophisticated misuses and rare use cases and tasks.

In this thesis we confirmed many assumptions, only supposed by cryptography researchers, by intuition, about how cryptography's complexity is underestimated and overlooked in software development. Based on our data and findings, we believe now that cryptographic software security needs a new generation of supporting technology (e.g., verification tools, APIs, and development frameworks), which is still to be shaped, being the subject of extensive research, worldwide. For instance, developers are driven by use cases when coding cryptographic software with crypto APIs. Thus, APIs could adapt to offer high-level services directly related to use cases. Also, cryptography experts will have to adapt to a new context where cryptographic technology needs to be bullet-proof and developer-friendly at the same time.

1.2.5 Future improvements

We foresee the following topics as **future improvements** to this thesis. First, more experimental evidence is needed to validate our current findings; particularly, by the time of writing, we did not perform any experiment (or quasi-experiment) with actual developers, in order to validate our findings against actual behavior. Second, we are aware that the current instance of our classification of cryptography misuse reflects what we have seen so far. Its maintenance and enhancement is a continuous task. Third, a tool to assist developers in building cryptographic software is in order, or, perhaps, a novel cryptographic API or framework to better encapsulate cryptographic services for developers.

1.2.6 Organization of the text

The next chapters are organized as follows. Chapter 2 is a verbatim compilation of published works grouped by research topics. Chapter 3 revisits our contributions and findings, placing them in the context of secure software engineering. Finally, Chapter 4 concludes the thesis with final remarks, expanding the discussion about future directions.

Chapter 2

Published works

This chapter is a verbatim compilation of published works organized by research topic. We arranged papers by research topic, instead of ordering them by chronology of publication, because readers are supposed to be more comfortable in reading a short sequence of papers related to the same topic, instead of traversing a long list of apparently unrelated papers organized in chronological order.

Next sections present published works grouped in three research topics as follows:

1. The empirical studies of crypto misuse in online communities (in Section2.1) features the mining of crypto misuse in coding forums (third publication [27]) and the longitudinal study on cryptography misuse over time (fifth publication [29]).

2. The evaluation of automated tools for cryptography (in Section2.2) features the literature survey on programming and verification of secure cryptographic software (first publication [25]) and the experimental evaluation of static code analysis tools for cryptography (fourth publication [31]).

3. The development methods for secure crypto software (in Section2.3) contains the methodology for development of secure cryptographic software (second publication [28]) and the systematic approach to cryptographic software security (sixth publication [30]).

2.1 Empirical studies in online communities

This section contains the following publications. First, the publication entitled *"Mining Cryptography Misuse in Online Forums"*. Second, the publication entitled *"A Longitudinal and Retrospective Study on How Developers Misuse Cryptography in Online Communities"*.

2.1.1 Mining Cryptography Misuse in Online Forums

This publication is entitled *"Mining Cryptography Misuse in Online Forums"* and was published at the IEEE International Conference on Software Quality, Reliability and Security (QRS 2016), in the International Workshop on Human and Social Aspect of Software Quality, held in the city of Vienna, Austria.

2016 IEEE International Conference on Software Quality, Reliability and Security Companion

Mining Cryptography Misuse in Online Forums

Alexandre Braga and Ricardo Dahab
Institute of Computing
State University of Campinas (UNICAMP)
Campinas, Brazil
ambraga@cpqd.com.br, rdahab@ic.unicamp.br

***Abstract*—This work analyzes cryptography misuse by software developers, from their contributions to online forums on cryptography-based security and cryptographic programming. We studied three popular forums: Oracle Java Cryptography, Google Android Developers, and Google Android Security Discussions. We applied a data mining technique, namely *Apriori*, to elicit association rules among cryptographic bad practices, platform-specific issues, cryptographic programming tasks, and cryptography-related use cases. We found that, with surprisingly high probabilities (90% for Java and 71% for Android), several types of cryptography misuse can be found in the posts, but unfortunately masked by technology-specific issues and programming concerns. We also found that cryptographic bad practices frequently occur in pairs or triples. We related triple associations to use cases and tasks, characterizing worst case scenarios of cryptography misuse. Finally, we observed that hard-to-use architectures confuse developers and contribute to perpetuate recurring errors in cryptographic programming.**

Keywords—cryptography misuse; Apriori algorithm; data mining; Java cryptographic architecture; secure coding

I. INTRODUCTION

The world is witnessing a notable increase in the use of security functions based on cryptographic technologies. Almost every day, ordinary developers execute common tasks related to cryptographic programming (e.g. encryption, signing, and key generation), in simple use cases historically associated to secure software (e.g. secure communication, user authentication, and file encryption). Intuitively, we see that secure software practitioners do recognize the importance of cryptography. However, in practice, they give little attention to its misuse when compared to the attention warranted to other programming vulnerabilities (e.g. buffer overflows and SQL injection), a state of affairs that contribute to developers' misconceptions when using cryptography in practice.

Thus, in this text, we aim at clarifying how cryptography has been misused by software developers and underestimated by secure software practitioners, when asking questions or giving advice about cryptographic coding in online forums.

The contributions of this work are the following. First, we devise a method to extract association rules among types of cryptographic misuse. As far as we know, this is the first attempt to use data mining techniques in order to learn association rules related to developer's misbehavior when programming cryptographic software. Second, we show evidence that several types of cryptography misuse found in posts from online forums (90% for Java and 71% for Android) have been masked by technology specific issues and programming concerns, usually preferred by developers. Third, we show that cryptographic bad practices also occur in pair or triple combinations, with high probabilities. Fourth, we identify worst case combinations of cryptography misuse, which are characterized by triple association rules for misuses related to security use cases and programming tasks.

This text is organized as follows. Section II gives background and related work. Section III details the research methodology. Section IV presents results. Section V discusses at length our findings. Section VI brings the conclusions.

II. BACKGROUND AND RELATED WORK

Lazar et al [1] have shown that 17% of cryptography vulnerabilities are inside software libraries, with the other 83% being misuses of those libraries. Also, Chatzikonstantinou et al [2] analyzed Android apps, concluding that about 88% of them showed some cryptography misuse.

According to Egele et al [3] and Shuai et al [4], the most common misuse is symmetric deterministic encryption, when a block cipher (e.g., AES or 3DES) uses Electronic Code Book (ECB) mode. There are cryptographic libraries in which ECB mode is the default option, automatically selected when the operation mode is not explicitly specified. A variation of this misuse is the asymmetric deterministic encryption with non-randomized RSA, identified by Gutmann [5].

Hardcoded Initialization Vector (IV) [3] is another frequent misuse. In several operation modes of block ciphers, IVs must be unique and unpredictable. The Counter (CTR) Mode requires unique IVs (without repetition). A related misuse is hardcoded seeds for Pseudorandom Number Generators (PRNGs) [3]. Still, other misuses arise when operation modes are exchanged without considering IV requirements. For instance, Java Cryptographic Architecture (JCA) [6] allow operation modes to be easily changed.

In mobile apps, libraries to handle SSL/TLS connections showed several issues when validating digital certificates. Georgiev et al [7] and Fahl et al [8] have shown that these libraries allow programmers to ignore parts of certificate validation in favor of usability or performance, but adding vulnerabilities. For instance, failures in signature verification or domain-name validation ease man-in-the-middle attacks.

Finally, Nadi et al [9] studied the obstacles developers face when using cryptography in Java, concluding that developers usually implement simple use cases (e.g. user authentication, login data storage, secure connections, and data encryption), but face difficulties when using low-level Java APIs. For instance, Shuai et al [10] discovered that password protection in Android is greatly affected by cryptography misuse.

978-1-5090-3713-1/16 $31.00 © 2016 IEEE
DOI 10.1109/QRS-C.2016.23

All these studies advanced the knowledge about cryptography misuse in secure software. However, they do not show how different kinds of cryptography misuse related to each other and are influenced by the target platform.

III. Research Methodology

This section details our methods of study and mining.

A. Study Method

The study included classification of cryptographic bad practices, post selection in targeted forums, and data analysis.

1) Classification of Cryptographic Bad Practices

We performed a literature review in order to identify cryptography bad practices and related advice aimed at secure software developers. We were not interested in advice for secure implementation of cryptographic algorithms. Instead, we looked for secure programming techniques for building cryptographic software. We searched through three sources: literature on software security that also covers cryptography issues [11]–[16], recent studies on cryptography misuse (in Section II), and industry initiatives for software security [17]–[20]. The resulting classification is shown in TABLE I.

TABLE I. Cryptography bad practices

Category	Bad practice type
Weak Cryptography (WC)	- Risky or broken cryptography - Proprietary cryptography - Deterministic symmetric encryption - Reversible or broken hash function - Custom implementation of standards
Poor Key Management (PKM)	- Short key, improper size, insufficient length - Hard-coded, static or constant keys - Hard-coded passwords for PBE - Reuse of keys with stream ciphers - Use of expired keys - Key distribution issues
Bad Randomness (BR)	- Use of statistic PRNGs - Predictable or low entropy seeds - Fixed, static or reused PRNG seeds
Program Design Flaws (PDF)	- Unsafe behavior or default - Insecure key handling (security of keys) - Insecure use of stream ciphers - Insecure combo of encryption and auth/hash - Side-Channel Attacks
Improper Certificate Validation (ICV)	- Missing validation of certificate - Broken SSL/TLS channel - Incomplete certificate validation (various) - Improper validation of hostname or user - Trust in certificates (wildcards, self signed)
Coding and Implementation Bugs (CIB)	- Password-based encryption (salt, count, hash) - Common coding errors (Various) - Bug in IV generation - No cryptography (e.g., NullCipher)
Cryptography Architecture and Infrastructure (CAI)	- Cryptography agility - API misunderstand, lacking knowledge or docs - Library or module (single point of access) - Randomness infrastructure - PKI and CA issues
Public-Key Cryptography (PKC)	- RSA weak keys - Insecure padding (no OAEP/PSS) for RSA - Deterministic asymmetric enc. (RSA/ECB) - Inadequate key length for RSA - Key agreement (e.g., DH) issues - Elliptic Curve Cryptography (ECC) issues
IV/Nonce Management (IVM)	- CBC mode with non-random IV - CTR mode with static counter - Fixed, hard-coded, or constant IV - Reusing nonce in encryption

There are nine main categories: weak cryptography (WC), poor key management (PKM), bad randomness (BR), program design flaws (PDF), improper certificate validation (ICV), coding and implementation bugs (CIB), cryptography architecture and infrastructure (CAI), public-key cryptography (PKC), and IV/nonce management (IVM). Many texts [11]–[16] already have their own classification. The bulk of work was then to merge them all, filling the gaps with recent works (from Section II) and industry concerns [17]–[20].

2) Targeted Forums and Posts Selection

We looked for programming forums possibly supported by experts in applied cryptography. Three forums were selected: *(i)* Oracle Java Cryptography (OJC) [21], a forum aimed at programming with Java Cryptographic Architecture (JCA); *(ii)* Google Android Developers (GAD) [22], a forum for Android programming; and *(iii)* Google Android Security Discussions (GASD) [23]. These public forums were chosen because their core technologies share the same Java-based API for cryptography, thus limiting the knowledge required by a code reviewer to the following aspects: Java programming, JCA, Android security, and applied cryptography.

Posts were collected in March 2016 and comprised a time period of five years, from January 2011 to December 2015. Posts were listed by date (newest first) and manually saved as PDF files. Also, metadata for each post (such as name, date of last activity, and number of views) were collected in a spreadsheet. In many forums, the default option for listings of posts is “sort by relevance”. However, relevance is context-sensitive, so we decided to use “sort by date” instead.

OJC is the most active forum, with the most posts in the selected time period. GAD and GASD are both very active in general, but showed less activity for cryptographic matters. For OJC, all 310 posts were collected, and the 155 most-viewed were selected for further analysis (50% of total). In both GAD and GASD, a pre-analysis showed that specific keywords, such as “encryption”, “hash” and “sign”, were covered by the more general keywords “cryptography” and “encryption”, which were used to select posts in GAD and GASD. For GAD, 170 posts were collected and the 100 most-viewed were selected for analysis. For GASD, 146 posts were collected and the 100 most-viewed were selected.

3) Data Analysis Method

The manual inspection (i.e., code review) was the method used to analyze each single post. Posts were inspected by a cryptography expert with the skills mentioned above. Before actually starting the analysis, he spent some time studying post style and structure and was able to identify a set of recurring topics related to environment and platform specific issues, in TABLE II. It also shows common programming tasks and simple use cases associated to cryptography in literature.

A misuse is a bad practice or a platform specific issue. Each post was inspected for occurrences of misuse, in various combinations. Each post was also categorized according to the main cryptography task carried out by its programmer, as well as the main security use case treated by that task. Many posts showed only discussions about threats or attacks, without a use case or programming task. When unable to identify the use case or task associated to a post, we discarded it from the data set. After discard, the OJC dataset was reduced to 140 posts, GAD to 71 posts, and GASD to 48 posts.

TABLE II. AUXILIARY CLASSIFICATION

Category	Type
Environment and Platform Specific Issues (EPSI)	- Configuration and installation issues - Key storage and recovery - Bug found or reported - Tool misuse or misunderstanding - Interoperation issues (e.g., platforms, versions, etc.) - Hardware integration issues
Cryptographic Programming Tasks	- Encryption and decryption (Enc) - Signature, MAC and verification (Sig) - Key generation or agreement (KG) - SSL/TLS secure channel (SSL) - Digital certificate handling (Cert)
Cryptographic Use Cases	- Encrypting Data at Rest (e.g., database, file, etc.) - Digital Rights Management - Secure Communication (VPN, SSL, HTTPS, etc.) - Password Protection and Encryption - Authenticate or Validate Data (e.g., sign, MAC)

This step resulted in a matrix of ones and zeros with lines representing posts and columns representing misuses (bad practices or platform issues), tasks, and use cases. A cell was marked with value one when a misuse (column) was found at the corresponding post (line). The same was done for tasks and use cases. A matrix was generated for each forum.

B. Data Mining Method

In order to find out interesting associations among bad practices and specific issues, we applied a data mining technique, the *Apriori* algorithm [24]. This technique has been traditionally applied in recommendation systems to suggest cross-selling of goods. Each matrix from the previous step was taken as a data set for the *Apriori* algorithm. Also, each post (line) counted as a transaction and each misuse (column) counted as an item present in that transaction. Data sets were preprocessed by summarizing misuse by categories.

In the *Apriori* algorithm, the support of an item (misuse) was computed as the percentage of the data set (posts) that contain that bad practice or issue. The support of an item set (set of more than one misuse) was calculated as the percentage of posts that contain this set. For instance, weak cryptography (WC) appeared in 26% of OJC posts, so this is WC's support. Also, weak cryptography appeared together with coding bugs (CIB) in 10% of OJC posts, so 10% is the support for this two-misuse set (WC&CIB), representing an interesting association rule between these two bad practices. This study adopted a minimum support of 2% to select useful association rules.

We also computed three supporting metrics in order to easily identify trustworthy rules: confidence, lift and leverage. The confidence of a rule was computed as the support of the misuse set divided by the support of the main (or leading) misuse of that set. A higher confidence indicated a trustworthy rule. Lift and leverage were used to exclude statistically independent (coincidental) associations [25][26].

Lift measures the number of times in excess of the expected, that items in a misuse set occur together, assuming they are statistically independent. Lift is computed as the support of a misuse set divided by the product of single supports for all misuses in that set. Lift is 1 if the items are statistically independent. A lift greater than 1 indicates a useful association rule; larger lifts strengthen that association.

Leverage measures the distance between the probabilities of items (in a set) that appear together and what would be expected if these items were statistically independent. Leverage is calculated as the support of an item set minus the product of single supports for all items in that set. Leverage is zero when items in the set are statistically independent. If the items have any kind of statistical dependence, the leverage would be greater than zero.

The evaluation of each forum proceeded by iteratively computing values for support, confidence, lift, and leverage for data sets consisting of single items (one bad practice or issue) and pairs of items, when the support of the pair was at least 2%. Triples were considered only when the supporting pair had at least a 5% support and the resulting triple showed a minimum support of 2%.

IV. RESULTS AND FINDINGS

This section presents general results and highlights interesting findings. First, we present the percentages of bad practices for all three forums together. Then, for each forum individually, we present association rules for pairs and triples of bad practices and issues, as well as for programming tasks and use cases. When suitable, we instantiate cases of misuse.

A. Single Occurrence of Cryptography Misuse

Single support for cryptography misuse (bad practices, platform specific issues), programming tasks, and use cases is summarized in TABLE III. The table shows that weak cryptography (WC) is the most common bad practice in both OJC and GAD. Also, all forums suffer negative influence from platform specific issues (EPSI).

Besides specific issues, OJC suffers the most influence from weak cryptography (WC, 26%), architectural issues (CAI, 20%), coding bugs (CIB, 17%), public key issues (PKC, 16%), and poor key management (PKM, 11%). This profile is probably due to API misuse, lack of knowledge in applied cryptography, and complexity of JCA.

TABLE III. *APRIORI* SINGLE SUPPORT

Misuses, Tasks and Use Cases	Single Support (%)		
Bad Practices and Issues (BP&I)	***OJC***	***GAD***	***GASD***
Weak Cryptography (WC)	**26%**	**21%**	**10%**
Poor Key Management (PKM)	**11%**	4%	2%
Bad Randomness (BR)	0%	1%	**10%**
Program Design Flaws (PDF)	6%	8%	**10%**
Improper Cert. Validation (ICV)	4%	3%	**15%**
Coding and Impl. Bugs (CIB)	**17%**	**17%**	6%
Crypto Architecture and Infra. (CAI)	**20%**	1%	**23%**
Pub. Key Crypto Issues (PKC)	**16%**	**10%**	4%
IV and Nonce Management (IVM)	5%	6%	0%
Env. and Plat. Specific Issues (EPSI)	**60%**	**32%**	**48%**
Crypto Programming Tasks (CPT)	***OJC***	***GAD***	***GASD***
Encryption and decryption	**26%**	**32%**	**21%**
Signature, MAC and verification	**16.5%**	**10%**	**10%**
Key generation or agreement	11%	4%	8.4%
SSL/TLS secure channel	8.5%	1.5%	8.3%
Digital certification	**17%**	1.5%	8.3%
Other or unidentified	21%	51%	44%
Crypto Use Cases (CUC)	***OJC***	***GAD***	***GASD***
Encrypting Data at Rest (EDR)	**31%**	**65%**	**54%**
Digital Rights Management (DRM)	0%	4.5%	2%
Secure Communication (SC)	**18%**	**11%**	**27%**
Password Protection and Enc. (PPE)	0%	**11%**	8.5%
Auth. or Validate Things (AVD)	**26.5%**	8.5%	8.5%

GAD suffers most from weak cryptography (WC, 21%), coding bugs (CIB, 17%), and public key issues (PKC, 10%). This profile is probably due to API misuse and lack of knowledge in cryptography programming. Despite preserving the Java API, Android has its own architecture for enabling cryptographic libraries, which simplifies installation and configuration, but brings new interoperational issues.

GASD suffers most from architectural issues (CAI, 23%), improper validation of certificates (ICV, 15%), and a mix of design flaws (PDF, 10%), bad randomness (BR, 10%) and weak cryptography (WC, 10%). Despite its focus on discussions about security issues, this forum frequently receives questions from developers about cryptographic programming. Over the years, a severe vulnerability found in Android's randomness infrastructure raised serious concerns in that community. Also, this forum is particularly concerned with vulnerabilities associated with certificate validation.

All three forums showed similar behavior for both programming tasks and use cases. The programming task most found in all three forums was encryption/decryption, followed by signing/verification. Interestingly, OJC also showed great interest in digital certification. The use case most found in all three forums was encrypting data at rest (EDR), followed by cryptographically secure communication (SC). OJC also showed great interest in data authentication (AVD).

B. Analysis of Association Rules for OJC

OJC presented association rules for cryptography misuse specific to Java. Relevant association rules with good values for confidence, lift and leverage are shown in TABLE IV. Rules are sorted by support value. For pair associations, architectural issues are strongly related to platform issues (CAI&EPSI, 12%) due to complexity of Java's cryptographic architecture. Posts within rule CAI&EPSI included accessing cryptographic hardware, configuration issues for hardware tokens, installation or configuration issues for libraries, key storage/recovery issues with external tools, and interoperation issues between Java and other languages.

Weak cryptography followed by coding bugs (WC&CIB, 6%) or public key issues (WC&PKC, 6%) are other easily recognized patterns. Posts within rule WC&CIB include uses of strings to hold encrypted data instead of byte arrays, wrong conversion of hexadecimal values to bytes, weak encryption with DES or MD5, and interoperation issues between Java and PHP. Posts within rule WC&PKC include uses of custom implementations of cryptographic algorithms, short keys for RSA, uses of SHA1 or MD5 hash functions, RSA without randomization, wrong length of RSA input, and interoperation issues between JCA and smartcards or Microsoft's cryptographic library and API (MSCAPI).

Mistakes in key management are also related to weak cryptography (WC&PKM, 4%), coding bugs (CIB&PKM, 4%), or public key issues (PKM&PKC, 5%). For example, posts within rule PKM&PKC include weak encryption with non-randomized RSA, 1024-bit (or shorter) keys for RSA, signing with RSA/MD5, coding proprietary key agreement, wrong length of RSA input, and interoperation issues between JCA and smartcards or MSCAPI.

Certificate validation suffers from platform deficiencies in handling and storing certificates (ICV&EPSI, 3%). Weak cryptography is also related to program design flaws when insecure defaults are adopted (WC&PDF, 3%). Mistakes in IV handling and design flaws (IVM&PDF, 2%) or key management (IVM&PKM, 2%) appear together when IVs and keys are hard-coded, and operation modes are omitted. Design flaws also relate to coding bugs (CIB&PDF, 2%) and key management (PDF&PKM, 2%).

For triple associations, we found occurrences of two patterns. First, the triple rule formed by weak cryptography, coding bugs, and design flaws (WC&CIB&PDF) showed up in 2% of posts. Posts related to rule WC&CIB&PDF include uses of 3DES in ECB mode, short keys for AES, and API's insecure default to AES/ECB. A common coding bug was the wrong conversion of ciphertext to String and vice-versa. Another triple rule formed by weak cryptography, coding bugs, and key management (WC&CIB&PKM) appeared in 2% of posts. Posts within rule WC&CIB&PKM include uses of 3DES in ECB mode, short keys for AES, and signing with RSA/SHA1 using 1024-bit (or shorter) key. Again, wrong conversion of keys to hexadecimal of String was common.

TABLE IV. ASSOCIATION RULES FOR OJC

Association Rule	Sup.	Conf.	Lift	Lev.
Pair Associations				
CAI&EPSI	12%	0.61	1.01	+0.00
WC&CIB	6%	0.24	1.42	0.02
WC&PKC	6%	0.22	1.32	0.01
PKM&PKC	5%	0.44	2.66	0.03
WC&PKM	4%	0.16	1.42	0.01
CIB&PKM	4%	0,21	1.82	0.02
ICV&EPSI	3%	0.67	1.11	0.59
WC&PDF	3%	0.11	1.68	0.01
IVM&PDF	2%	0.43	6.67	0.02
IVM&PKM	2%	0.43	3.75	0.02
CIB&PDF	2%	0.13	1.94	0.01
PDF&PKM	2%	0.33	2.92	0.01
Triple Associations				
WC&CIB&PDF	2.14%	0.33	5.19	0.02
WC&CIB&PKM	2.14%	0.33	2.92	0.01

TABLE V. BAD PRACTICES AND ISSUES BY TASK IN OJC

BP&I	Crypto Programming Tasks				
	Enc	*Sig*	*KG*	*SSL*	*Cert*
WC	**11%**	6%	-	3%	-
PKM	6%	-	3%	-	-
PDF	4%	-	1%	1%	-
ICV	-	-	1%	1%	2%
CIB	**9%**	-	2%	-	-
CAI	-	-	-	-	**9%**
PKC	5%	5%	4%	-	-
IVM	4%	-	1%	-	-
EPSI	-	**12%**	-	5%	**11%**

TABLE VI. BAD PRACTICES AND ISSUES BY USE CASES IN OJC

BP&I	Crypto Use Cases				
	EDR	*DRM*	*SC*	*PPE*	*AVD*
WC	**12%**	-	5%	-	8%
PKM	7%	-	-	-	-
PDF	5%	-	-	-	-
ICV	-	-	4%	-	-
CIB	**10%**	-	-	-	-
CAI	-	-	-	-	5%
PKC	6%	-	4%	-	6%
IVM	4%	-	-	-	-
EPSI	-	-	-	-	**18%**

Misuses associated to programming tasks in OJC are in TABLE V. Most bad practices are associated to encryption (Enc), followed by key generation (KG) and SSL secure channels. Bad randomness (BR) did not show up in any programming task for this forum. Encryption and key generation do not suffer with Java specific issues. On the other hand, signing (Sig) and digital certification (Cert) are frequently affected by platform specific issues.

Bad practices and issues associated to use cases in OJC are in TABLE VI. Most bad practices are associated to encrypting data at rest (EDR), with attention to weak cryptography (WC) and coding bugs (CIB). Authenticating or validating data (AVD) is affected by mistakes specific to Java. Secure communication (SC) is affected by mistakes in certificate validation and (weak) public-key cryptography. In OJC, nobody mentioned password protection or DRM, probably because there is no direct need for these use cases in Java.

Finally, triple rule WC&CIB&PDF is related to encryption task in 2% of posts, which characterizes a worst case scenario for this task. Also, triple rule WC&CIB&PDF relates to EDR use case in 2% of posts, again, typifying this worst case.

C. Analysis of Association rules for GAD

GAD presented a great number of association rules, when compared to OJC and GASD. Relevant rules sorted by support value are shown in TABLE VII. For pair associations, the most prevalent rules include weak cryptography associated to coding bugs (WC&CIB, 10%) or platform specific issues (WC&EPSI, 10%). Posts within rule WC&CIB are mostly related to Password-Based Encryption (PBE) and include uses of MD5 or SHA1, block ciphers in ECB mode, fixed IVs, and PBE issues (e.g., small counts or fixed salt), including keys directly derived from password hashes. Posts within rule WC&EPSI include various misuse of weak cryptography (MD5, SHA1, DES, and ECB), interoperation issues between computers and Android devices, faulty SSL implementations (in Android 2.2), errors in converting data to String, and backward incompatibility with older versions.

Also, Android specific issues are related to coding bugs (CIB&EPSI, 8%), public-key issues (PKC&EPSI, 7%) and design flaws (PDF&EPSI, 6%). Posts within rule CIB&EPSI include backward compatibility issues in Android 4.4, errors in decrypting for different versions of Android, various PBE issues (e.g., no KDF, fixed salt, and small count), and improper conversion of (cipher text) byte array to String. Posts within rule PKC&EPSI include interoperation issues when exchanging digital signatures between dotNET and Java, use of deterministic RSA, SSL/ECC not working in Android 2.2, and errors when recovering RSA keys from Android keystore.

Posts related to rule IVM&CIB (6%) are affected by fixed IVs when encrypting user generated data with PBE, confusion between salt and IVs in PBE, deriving keys directly from hashing passwords with MD5, and errors in converting integer to byte and back to integer before decryption.

For triple associations, we found four patterns, all of them affected by Android specific issues, and one of which with a relatively high value. Weak cryptography with Android specific issues and coding bugs (WC&EPSI&CIB) counted for 5.6% of posts and was the triple rule with the highest support found in this study. Posts related to rule WC&EPSI&CIB are affected by uses of DES and MD5, PBE with small count and fixed salts, wrong conversion of cipher text to String, deriving keys directly from password hash (MD5), and interoperation issues when exchanging ciphertexts among Android versions.

Posts within rule WC&EPSI&PKC include deterministic encryption with RSA, custom implementation of SSL, and interoperation issues with SSL. Posts within triple rule CIB&EPSI&IVM include PBE with fixed salts, improper conversion of ciphertext to String, deriving keys directly from hashing a password with MD5, and interoperation issues when exchanging ciphertexts among Android versions, as well as between a computer and an Android device. Posts within triple rule PDF&EPSI&CIB include deriving keys directly from hashing passwords, insecure defaults (AES/ECB), and errors in exchanging ciphertexts among versions of Android.

TABLE VII. ASSOCIATION RULES FOR GAD

Association Rule	Sup.	Conf.	Lift	Lev.
Pair Associations				
WC&CIB	10%	0.47	2.76	0.06
WC&EPSI	10%	0.47	1.10	0.01
CIB&EPSI	8%	0.50	1.18	0.01
PKC&EPSI	7%	0.71	1.69	0.03
IVM&CIB	6%	1.00	5.92	0.05
PDF&EPSI	6%	0.67	1.58	0.02
PDF&PKC	4%	0.50	5.07	0.03
WC&PKC	3%	0.13	1.35	0.01
IVM&EPSI	3%	0.5	1.18	+0.00
CIB&PDF	3%	0.17	1.97	0.01
ICV&EPSI	3%	1.00	2.37	0.42
PDF&PKM	3%	0.33	7.89	0.02
PKM&PKC	3%	0.67	6.76	0.02
Triple associations				
WC&EPSI&CIB	5.6%	0.56	3.33	0.04
WC&EPSI&PKC	2.8%	0.28	2.86	0.02
CIB&EPSI&IVM	2.8%	0.35	6.25	0.02
PDF&EPSI&CIB	2.8%	0.47	2.78	0.02

TABLE VIII. BAD PRACTICES AND ISSUES BY TASK IN GAD

BP&I	Crypto Programming Tasks				
	Enc	*Sig*	*KG*	*SSL*	*Cert*
WC	**13%**	6%	-	-	-
PKM	4%	-	-	-	-
BR	1%	-	-	-	-
PDF	6%	1%	-	-	1%
ICV	-	-	1%	-	1%
CIB	**11%**	4%	1%	-	-
CAI	1%	-	-	-	-
PKC	4%	3%	1%	-	1%
IVM	3%	1%	1%	-	-
EPSI	-	6%	3%	-	1%

TABLE IX. BAD PRACTICES AND ISSUES BY USE CASE IN GAD

BP&I	Crypto Use Cases				
	EDR	*DRM*	*SC*	*PPE*	*AVD*
WC	-	3%	3%	7%	-
PKM	4%	-	-	-	-
BR	1%	-	-	-	-
PDF	8%	-	-	-	-
ICV	3%	-	-	-	-
CIB	-	-	-	7%	-
CAI	1%	-	-	-	-
PKC	-	-	3%	-	1%
IVM	6%	-	-	-	-
EPSI	-	-	6%	-	7%

Bad practices and issues associated to programming tasks in GAD are in TABLE VIII. In this Android forum, most bad practices affect encryption, and weak cryptography and coding bugs are the most perceived mistakes. Signing is moderately affected by bad practices. Key generation and certificate handling are tasks scarcely affected. Programming of SSL channels did not show any relevant bad practice. Also, we found three pair rules related to encryption: WC&CIB (5.6%), WC&EPSI (7%), and CIB&EPSI (7%). Finally, encryption is related to rule WC&EPSI&CIB in 4.2% of posts, which typifies a worst case for this forum.

Bad practices and issues associated to main use cases in GAD are in TABLE IX. Most bad practices are associated to encrypting data at rest (EDR) with special attention to design flaws and IV/nonce management (IVM). EDR is associated to two rules in 5.6 % of posts each: CIB&IVNI and PDF&EPSI. Secure communication (SC), password protection (PPE), and authentication and validation of data (AVD) are moderately affected by bad practices. Secure communication is negatively affected by complexity of Android's certificate storage. Interestingly, PPE use case is related to rule WC&CIB in 7% of posts and to rule WC&EPSI&CIB in 4.2 % of posts, characterizing a worst case misuse.

D. Analysis of Association Rules for GASD

This forum presented the shortest rule set. The relevant association rules sorted by support value are in TABLE X. For pair associations, bad randomness is one of the most prevalent rules and is associated to architectural (BR&CAI, 6%) and platform specific (BR&EPSI, 4%) issues. Posts within rule BR&CAI discussed the influence of bad randomness provided by the Android platform for generation of cryptographic keys.

Design flaws and architecture issues are related to discussions of specific topics (PDF&CAI, 4%) with no source code. Posts within rule PDF&CAI include bugs in Android's key storage and errors when accessing security modules. Four low-value (2%) rules show that weak cryptography is related to coding bugs (WC&CIB), improper certificate validation (WC&ICV), design flaws (WC&PDF), and poor key management (WC&PKM). Finally, public key issues are related to design flaws (PDF&PKC) or platform issues (PKC&EPSI). No relevant triple rule was found.

Bad practices and issues associated to programming tasks in GASD are shown in TABLE XI. Signing and key generation are the most negatively affected tasks of this forum, being associated to six and five bad practices, respectively. In particular, key generation in Android is greatly affected by platform specific issues concerning randomness, influencing architecture of apps. Certificate handling and secure communication are moderately affected in general, but special attention should be given to improper certificate validation in both tasks. Encryption is affected only by platform specific issues, but with a high percentage. This happened due to interoperation issues among different versions of Android or server-side software. IV and nonce management do not show up with any programming task.

Bad practices and issues associated to main use cases for GASD are in TABLE XII. In this forum, most bad practices are associated to authentication/validation of data, with a moderate focus in architecture issues. Encrypting data at rest is affected by bad randomness, design flaws, and architectural issues. Particular attention should be given to the negative influence of platform specific issues over encryption data at rest. IV and nonce management did not show up with any use case. DRM concerns appeared only in high-level discussions, so no bad practice was bound to it.

TABLE X. ASSOCIATION RULES FOR GASD

Association Rule	Sup.	Conf.	Lift	Lev.
Pair Associations				
BR&CAI	6%	0.6	2.62	0.04
BR&EPSI	4%	0.4	1.07	+0.00
PDF&CAI	4%	0.4	1.75	0.02
WC&CIB	2%	0.2	3.20	0.01
WC&ICV	2%	0.2	1.37	0.01
WC&PDF	2%	0.2	1.92	0.01
WC&PKM	2%	0.2	9.60	0.02
PDF&PKC	2%	0.2	4.80	0.02
PKC&EPSI	2%	0.5	1.33	0.01

TABLE XI. BAD PRACTICES AND ISSUES BY TASK IN GASD

BP&I	Crypto Programming Tasks				
	Enc	*Sig*	*KG*	*SC*	*Cert*
WC	-	2%	-	-	-
PKM	-	2%	-	-	-
BR	-	-	4%	-	-
PDF	-	2%	-	-	
ICV	-	4%	-	4%	6%
CIB	-	2%	2%	-	-
CAI	-	-	4%	2%	2%
PKC	-	2%	2%	-	-
EPSI	10%	-	4%	-	4%

TABLE XII. BAD PRACTICES AND ISSUES BY USE CASE IN GASD

BP&I	Crypto Use Cases				
	EDR	*DRM*	*SC*	*PPE*	*AVT*
WC	-	-	-	-	2%
BR	6%	-	-	-	2%
PDF	8%	-	-	-	2%
ICV	-	-	13%	-	-
CIB	-	-	-	2%	2%
CAI	13%	-	-	-	4%
PKC	-	-	-	-	2%
EPSI	21%	-	10%	-	-

V. DISCUSSION OF FINDINGS

Our previous experience in developing cryptographic software provided the necessary skills and perspective to carry out this study. However, it could also infuse our results with a confirmation bias for all the mistakes we have seen in practice. We believe that by targeting three different forums, we were able to minimize this confirmation bias, because bad practices showed up in specific realizations for each forum, forcing us to focus on details otherwise neglected by the accustomed eye.

We have noticed that inherently complex, hard-to-use architectures distract developers from actual cryptographic bad practices and contribute to perpetuate recurring errors in cryptographic programming. For instance, one curious reason developers gave to use homemade code for cryptographic algorithms (one kind of weak cryptography) is to avoid dependencies to external libraries.

In programming forums, the number of posts showing any kind of misuse is consistent with statistics found in related work. For instance, OJC showed 90% of posts with misuse. GAD presented 71% of posts with misuse. GASD showed only 48%, but this forum is not for programming only. Other research did not divide misuses in bad practices and specific issues, as we did. Posts showing only bad practices counted for 71% in OJC, 48% in GAD, and 56% in GASD. Also, the small size of GASD's data set may lead to false association rules due to coincidence. However, findings were confirmed by the other two forums.

A. For Java Developers

Developers frequently showed little concern to security in general and were more interested in coding tasks. Java cryptographic architecture presents issues with installation and configuration that diverts developers from actual tasks of cryptographic programming. Also, specific issues showed up when integrating Java programs to cryptographic hardware or communicating with applications in other platforms. These troublesome issues frequently obfuscate bad practices in the same code. For instance, in Java, the worst case misuse occurs when developers write buggy code for encrypting data and use weak cryptography by accidentally adopting insecure defaults. This suggests that developers lack programming skills in general, and knowledge of Java's cryptographic API in particular.

```
01 byte[] pt = ("OJC worst case..").getBytes();
02 KeyGenerator g=KeyGenerator.getInstance("AES");
03 g.init(128); Key k = g.generateKey();
04 // encryption
05 Cipher e = Cipher.getInstance("AES");
06 e.init(Cipher.ENCRYPT_MODE, k);
07 byte[] ct = e.doFinal(pt);
08 String s = new String(ct);
09 // decryption in other machine or platform
10 Cipher d = Cipher.getInstance("AES");
11 d.init(Cipher.DECRYPT_MODE, k);
12 byte[] pt2 = d.doFinal(s.getBytes());
```

Fig. 1. Example of Java's worst case misuse (WC&CIB&PDF).

For the non-expert in applied cryptography, the code fragment in Figure 1 illustrates a realization of Java's worst case scenario with rule WC&CIB&PDF. This code snippet shows that plaintext is directly obtained from a String object (line 01) and weak cryptography is accidentally used (line 05) when the Cipher object for AES defaults to ECB mode, due to a program design flaw of not explicitly choosing the operation mode. Then, a coding bug occurs when encrypted data is saved to a String object (line 08), making it dependent of character encodings, and retrieved for decryption (line 12), with unexpected results.

This defect may manifest itself when the code is executed by different applications or distinct Java Virtual Machines (JVMs) with distinct character encodings. This way, a developer would ask for help concerning interoperation issues, maybe ignoring other cryptographic misuse.

B. For Android Developers

Android solved many of the installation and configuration problems faced by Java developers when doing cryptographic programming. On the other hand, this mobile platform brings to daily troubleshooting a variety of interoperation issues brought by the diversity of hardware and software available in that platform. Several less than 5% rules suggest that Android developers are more prone to use cryptography without proper education. Again, developers get confused by troublesome interoperation and distracted from actual cryptographic pitfalls in the code. Android developers show specific need for DRM, but homemade code leads to weak cryptography. Also, encryption of locally stored passwords is a common need when accessing remote services, but frequently misused. For instance, a worst-case scenario occurs when developers suffer with interoperation issues among devices, but in fact use weak cryptography to protect stored passwords, and derive keys directly from password hashes.

```
01 // weak hash of user's password
02 md = MessageDigest.getInstance("MD5");
03 byte[] hash = md.digest(password.getBytes());
04 // weak PRNG with fixed seed
05 sr = SecureRandom.getInstance("SHA1PRNG");
06 sr.setSeed(hash.getBytes());
07 byte[] keyBytes = new byte[16];
08 sr.nextBytes(keyBytes);
09 // crypto key derived from password
10 ks = new SecretKeySpec(keyBytes,"AES");
```

Fig. 2. Example of Android's worst case misuse (WC&EPSI&CIB).

Again, for the non-expert in cryptography, the code snippet in Figure 2 illustrates a realization of Android's worst case misuse with rule WC&EPSI&CIB. This code snippet shows that weak cryptography occurs when MD5 is directly used to hash the byte representation of a String object containing a password (lines 02 and 03). Then, in a mistaken implementation of Password-Based Encryption, the hash value feeds a PRNG with a fixed seed (lines 05 and 06) in order to always generate the same bytes (line 08), from which a key is derived (line 10). A platform specific issue, due to a backward incompatible bug fix, may cause SHA1PRNG to behave differently in distinct versions of Android, resulting in the generation of different keys.

This defect may manifest itself in two distinct ways. One, when the user updates his/her device's operating system to a new version. Another, when the user changes his password to one containing special characters. In this case, the developer would ask for help concerning interoperation issues and eventually discover a report about a bug fix for a specific version. As a result, the policy for password security would be updated to exclude special characters. However, in order to keep backward compatibility, the custom PBE code would never be changed.

C. For Experts in Applied Cryptography

This study revealed that bad practices and issues appear in different layers of programming activity. Platform specific issues dominate architectural concerns and obfuscate cryptography bad practices. Two shallow issues are in the surface of programs and could be easily found by skilled developers: weak cryptography and coding bugs. When shallow issues are not present, programmers have to look under the surface of programs for complex issues like improper certificate validation, bad randomness, and public key cryptography issues (mostly related to RSA). Avoiding these bad practices usually require more knowledgeable

developers or greater support from experts. Then, developers have to dive deeper into system design and face the most challenging problems, such as program design flaws, poor key management, and IV/nonce management. This array of misuses suggests that novice or unskilled developers are not aware of key management needs and other secure design challenges. It is not surprising that these deeper issues appear less frequently in programming forums. These bad practices require a change in system design in order to be tackled, and must be supported by cryptography experts.

When looking deeper into misuse categories, sometimes absences were more interesting than findings. We did not see, in programming forums (OJC and GAD), posts concerned with insecure combination of encryption and authentication or side-channel vulnerabilities. This suggests that developers are not aware of design flaws for authenticated encryption or padding oracle attacks. Another remarkable absence was the concern with weak keys in public-key cryptography (e.g., RSA, DH), suggesting that developers are taking for granted parameters generated by (faulty) tools. Also, misuse of PRNGs was barely mentioned in these forums, suggesting that developers have no doubts about Java's SecureRandom API.

On the other hand, in GASD, posts within rule BR&EPSI are related to discussions about a severe vulnerability found in Android's PRNG and interoperation issues among versions of Android, when using a PRNG from an open-source library. Discussions about entropy gathering and PRNGs internals by security experts caused this pattern. Also, this forum showed discussions about platform specific issues (EPSI) related to encrypting data at rest (EDR). This was caused by developers' misunderstandings about the internal works of Android's storage encryption and its influence on app-level encryption.

Finally, the use of data mining techniques to analyze developer's coding mistakes in cryptographic programming is promising, but requires further automation, in order to scale to bigger data sets. In particular, code inspection is highly dependent on reviewer's experience and was performed manually, but could benefit from customized tools for static analysis of cryptographic code.

VI. Concluding Remarks

This text presented a study about cryptography misuse by software developers when contributing to online forums specialized in cryptography-based security and cryptographic programming. By associating cryptography bad practices to platform specific issues, in three different forums, we were able to identify recurring patterns of cryptography misuse, including worst cases scenarios when at least three misuses appear together, in the same piece of code, with a relatively high probability.

Finally, this study is a step forward in better understanding pitfalls in cryptographic programming and could contribute to building the next generation of architectures, methods, and tools for the development of secure cryptographic software. In the short term, we foresee the use of association rules in metrics for measuring coding quality. In the long run, we devise misuse-aware APIs and misuse-resistant architectures.

Acknowledgment

Alexandre Braga thanks CNPq and Intel for the financial support. Ricardo Dahab thanks FAPESP, CNPq, CAPES, and Intel for partially supporting this work.

References

[1] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why Does Cryptographic Software Fail?: A Case Study and Open Problems," in 5th Asia-Pacific Workshop on Systems, 2014, pp. 7:1–7:7.

[2] A. Chatzikonstantinou, C. Ntantogian, C. Xenakis, and G. Karopoulos, "Evaluation of Cryptography Usage in Android Applications," in 9th EAI Int. Conf. on Bio-inspired Information and Comm. Tech., 2015.

[3] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," ACM SIGSAC conference on Computer & communications security, pp. 73–84, 2013.

[4] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications," in IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC), 2014, pp. 75–80.

[5] P. Gutmann, "Lessons Learned in Implementing and Deploying Crypto Software," Usenix Security Symposium, 2002.

[6] "Java Cryptography Architecture" [Online]. Available: docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html.

[7] M. Georgiev, S. Iyengar, and S. Jana, "The most dangerous code in the world: validating SSL certificates in non-browser software," in Proc. of the ACM Conf. on Computer and Comm. Security, 2012, pp. 38–49.

[8] S. Fahl, M. Harbach, and T. Muders, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," in ACM conference on Computer and communications security, 2012, pp. 50–61.

[9] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "'Jumping Through Hoops': Why do Java Developers Struggle With Cryptography APIs?," in The 38th International Conference on Software Engineering, 2016.

[10] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Analysis on Password Protection in Android Applications," 9th Int. Conf. on P2P, Parallel, Grid, Cloud and Internet Computing, 2014, pp. 504–507.

[11] J. Viega and G. McGraw, Building Secure Software. 2001.

[12] M. Howard and D. LeBlanc, Writing Secure Code, 2003.

[13] B. Chess and J. West, Secure Programming with Static Analysis, 2007.

[14] M. Howard, D. LeBlanc, and J. Viega, 24 Deadly Sins of Software Security, 2009.

[15] M. Howard and S. Lipner, The Security Development Lifecycle, 2006.

[16] A. Shostack, Threat Modeling: Designing for Security, 2014.

[17] "Avoiding The Top 10 Software Security Design Flaws," IEEE Cybersecurity Initiative (CYBSI), 2014. [Online]. Available: http://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf.

[18] "Fundamental Practices for Secure Software Development," Safecode, 2011. [Online]. Available: http://www.safecode.org/wp-content/uploads/2014/09/SAFECode_Dev_Practices0211.pdf.

[19] "OWASP Testing Project v4," OWASP, 2015. [Online]. Available: https://www.owasp.org/index.php/OWASP_Testing_Project.

[20] "TOP 25 Most Dangerous Software Errors," SANS/CWE. [Online]. Available: www.sans.org/top25-software-errors.

[21] "Oracle Java Cryptography." [Online]. Available: https://community.oracle.com/community/java/java_security/cryptography.

[22] "Google Android Developers." [Online]. Available: https://groups.google.com/forum/#!forum/android-developers.

[23] "Android Security Discussions." [Online]. Available: https://groups.google.com/forum/#!forum/android-security-discuss.

[24] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," in Proceedings of the 20th International Conference on Very Large Data Bases, 1994, pp. 487–499.

[25] M. R. Murthy, Introduction to Data Mining and Soft Computing Techniques, 2015.

[26] EMC Education Services, Data Science and Big Data Analytics: Discovering, Analyzing, Visualizing and Presenting Data, 2015.

2.1.2 A Longitudinal and Retrospective Study on How Developers Misuse Cryptography in Online Communities

This publication is entitled *"A Longitudinal and Retrospective Study on How Developers Misuse Cryptography in Online Communities"* and was published at XVII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg 2017), held in the city of Brasília, Brazil.

A Longitudinal and Retrospective Study on How Developers Misuse Cryptography in Online Communities

Alexandre Braga[1,2], **Ricardo Dahab**[2]

[1] Fundação CPqD Centro de Pesquisa e Desenvolvimento em Telecomunicações
R. Dr. Ricardo Benetton Martins, 1.000, Parque II do Polo de Alta Tecnologia
Campinas, SP, Brazil, Zip Code 13086-510

[2]Institute of Computing, State University of Campinas
Av. Albert Einstein, 1251, Cidade Universitária Zeferino Vaz
Campinas, SP, Brazil, Zip Code 13083-852

ambraga@cpqd.com.br,rdahab@ic.unicamp.br

Abstract. *Software developers participating in online communities benefit from quick solutions to technology specific issues and, eventually, get better in troubleshooting technology malfunctioning. In this work, we investigate whether developers who are part of online communities for cryptography programming are getting better in using cryptography with time. This is a crucial issue nowadays, when "real-world crypto" is becoming a topic of serious investigation, not only academically but in security management as a whole: cryptographic programming handled by non-specialists is an important and often invisible source of vulnerabilities [RWC]. We performed a retrospective and longitudinal study, tracking developers' answers about cryptography programming in two online communities. We found that cryptography misuse is not only common in online communities, but also recurrent in developer's discussions, suggesting that developers can learn how to use crypto APIs without actually learning cryptography. In fact, we could not identify significant improvements in cryptography learning in many daily tasks such as avoiding obsolete cryptography. We conclude that the most active users of online communities for cryptography APIs are not learning the tricky details of applied cryptography, a quite worrisome state of affairs.*

Resumo. *Desenvolvedores de software, participantes de comunidades on-line, costumam se beneficiar de soluções rápidas para problemas tecnológicos e, eventualmente, melhoram suas habilidades na resolução de problemas de mau funcionamento da tecnologia. Neste trabalho, investigamos se desenvolvedores de software criptográfico que participam de comunidades on-line se tornam melhores no uso de criptografia com o tempo. Esse é um aspecto de segurança crucial nos dias de hoje, em que "real-world crypto" se tornou um tópico de interesse sério, não só academicamente, mas no gerenciamento de segurança como um todo: programação criptográfica feita por não-especialistas é uma fonte frequente e muitas vezes invisivel de vulnerabilidades [RWC]. Realizamos um estudo retrospectivo e longitudinal para rastrear as respostas dos desenvolvedores sobre programação de criptografia em duas comunidades on-line. Descobrimos que o uso indevido da criptografia é não apenas comum em comunidades on-line, mas também é recorrente nas discussões, sugerindo que os desenvolvedores aprendem a usar as APIs criptográficas sem realmente aprender criptografia. Não conseguimos identificar melhora alguma na percepção e aprendizado de vulnerabilidades criptográficas, mesmo em tarefas simples como a de evitar o uso de criptografia obsoleta. Concluimos, assim, que os usuários ativos nas comunidades on-line para APIs criptográficas não tem evoluído no seu aprendizado dos detalhes e armadilhas do uso da criptograpfia, um estado de coisas muito preocupante.*

1. Introduction

Software developers are regular users of security mechanisms (e.g., security APIs, protocols, and tools), but are, by no means, security experts. They, however, do make security decisions that have a huge impact on end-user and system security. Developers are also frequent users of online communities for programming. The agility in problem solving provided by many question-and-answer communities brings benefits to ordinary programmers lacking knowledge in specific topics, such as secure coding. In this work, we investigate whether developers participating in online communities for cryptography programming are getting better in using cryptography with time. Also, we investigate whether cryptography misuse is persistent in posts of specific developers.

Cryptography misuse is a programming bad practice frequently found in misuse cases of cryptographic software, ultimately leading to vulnerabilities, but also associated to design flaws and insecure architectural choices [Braga and Dahab 2016]. Improvement in cryptography knowledge can be evidenced by a time series showing a steady decrease in the number of cryptography misuses. Also, persistence of a specific cryptography misuse can be illustrated by the repetition by developers of the same misuse over and over.

We tracked users (i.e. developers) of two online communities for programming, along with their questions and answers, from posts in a data set provided by a previous study [Braga and Dahab 2016]. We recorded the occurrence of known cryptography misuses for selected developers over a period of five years. We then computed statistics of cryptography misuse associated to specific moments in time for each tracked developer.

We found that the use of weak cryptography (e.g., broken algorithms or misconfigured implementations of standards) is not only common in online communities, but also recurrent in developers' discussions, suggesting that they learn how to use crypto APIs without actually learning cryptography. We also found that the lack of knowledge in cryptography is a recurrent source of coding bugs in API usage and does not depend on how long developers use cryptography APIs. We observed that platform issues dominate design concerns and obfuscate many complex cryptography misuses, which go unnoticed by developers in long lifespans. In summary, we conclude that users of online communities are not actually learning cryptography, despite their immediate gains in solving current programming issues related to cryptographic APIs.

This study is longitudinal, i.e., it performed repeated observations of the same developers over a period of time. Also, it is retrospective because it looks back in time using existing data. As far as we know, this is the first such study of cryptography misuse in online communities. The main contributions of this work are the following: *(i)* a method for clustering developers' posts from their asynchronous lifespans in online communities; *(ii)* a longitudinal study of selected developers of two communities, showing similarities and differences of these communities concerning how developers misuse cryptography; *(iii)* evidence that cryptography misuse is persistent across communities and developer lifespans; and *(iv)* evidence that developers learn how to use cryptography APIs without learning cryptography.

This text is organized as follows. Section 2 analyses related work and Section 3 details our research method. Section 4 explains our results and findings, while Section 5 details two users' lifespans. Section 6 discusses our findings and Section 7 shows our conclusions.

2. Related Work

[Fahl et al. 2012] investigated the SSL/TLS protocol usage in Android apps from Google Play and discovered security threats posed by misuses of that protocol. [Egele et al. 2013] were among the first to perform large-scale experiments in Google Play App Store to measure cryptographic

misuse in Android with the standard Crypto API. Their main contribution was a broad view of the prevalence of misused cryptographic functionality in Android apps. These two pioneering works were followed by others on related topics (e.g., [Lazar et al. 2014, Shuai et al. 2014, Georgiev et al. 2012, Chatzikonstantinou et al. 2015]).

[Wang and Godfrey 2013] were among the first to analyze API-related posts from a questions-and-answers (Q&A) website for mobile app development. They discovered repetitive scenarios with obstacles in API usage to developers, not specifically related to security. In a recent work [Wang et al. 2015], they investigated methods and proposed a methodology to distill and rank Q&A posts with API-related issues that would be valuable to API designers.

[Nadi et al. 2016] performed an empirical investigation into the obstacles developers face while using the Java cryptography APIs and the programming tasks they perform (e.g., authenticate users, store login data, establish secure connections, and encrypt data). By triangulating data from Stack Overflow posts, GitHub repositories, and developers' surveys, they found that developers find it difficult to use cryptographic algorithms correctly, despite being confident with cryptography concepts. They also found that cryptographic APIs are generally perceived as too low-level and not task-oriented.

[Acar et al. 2016b, Acar et al. 2016a] systematically analyzed the impact to code security of information resources commonly used by developers. They surveyed app developers who have published in the Google Play market, conducted a lab study with Android developers, analyzed 139 Stack Overflow threads accessed by developers during the lab study, and statically analyzed a random sample of Google Play apps. They concluded that real-world developers use Q&A communities as a major resource for solving programming problems, including security problems, suggesting that those online communities help developers to arrive at functional solutions more quickly than other resources. However, because online communities contain many insecure answers, developers who rely on this resource are likely to create less secure code. Also, access to quick solutions via a Q&A community may also inhibit developers' security thinking or reduce their focus on security.

[Braga and Dahab 2016] performed a transversal study to analyze how developers misuse cryptography in two online communities: Oracle Java Cryptography (OJC) and Google Android Developers (GAD). That work showed not only the most frequent cryptography misuses (e.g., weak cryptography, coding bugs, etc.), but also relationships among misuses through strong associations of double or triple misuses that appear together with non-negligible probabilities.

Most of the above-mentioned studies focus on the same online community or app store, with little variation. Also, none of these works study the behavior of frequent users over time, in order to examine whether developers are getting better in using cryptography with time.

3. Methodology

We analyzed data collected by a previous study [Braga and Dahab 2016] and observed that repeated measures were made for some developers. This fact motivated us to perform a retrospective, longitudinal study to analyze developers' behavior from a series of observations already made about them. This study is longitudinal because it performed repeated observations of the same developers (and their posts) over a period of time. Also, it is retrospective because it looks back in time using existing data.

Roughly speaking, our method segments the set of posts (collected for specific developers with determined lifespans) into a predefined number of clusters. When ordered chronologically, these clusters determine the phases a developer is supposed to pass for learning cryptography. These phases are then analyzed for the occurrence of cryptography misuses which are well-known in secure software development and secure coding.

The following subsections detail our methodology in four topics: classification of cryptography misuse, selection of communities and posts, selection of developers to evaluate, and method for clustering posts from a developer's lifespan.

3.1. Classification of Cryptography Misuse

The original study [Braga and Dahab 2016] introduced a classification of cryptography misuses in order to capture how software developers actually misuse cryptography in practice. The classification has nine categories: Weak Cryptography (WC), Bad Randomness (BR), Coding and Implementation Bugs (CIB), Program Design Flaws (PDF), Improper Certificate Validation (ICV), Public-Key Cryptography (PKC) issues, Poor Key Management (PKM), Cryptography Architecture Issues (CAI), and IV/Nonce Management (IVM) issues. Table 1 details categories in descriptive subsets.

The classification collected cryptography misuse from various sources, including software security books (e.g., [Viega and McGraw 2001, Howard and LeBlanc 2003, Chess and West 2007, Howard et al. 2009, Howard and Lipner 2006, Shostack 2014]), studies on cryptography misuse (e.g, [Lazar et al. 2014, Chatzikonstantinou et al. 2015, Egele et al. 2013, Shuai et al. 2014, Braga and Dahab 2015, Georgiev et al. 2012, Fahl et al. 2012]), newly discovered misuses (e.g., [Alashwali 2013, Bos et al. 2014, Mart and Hern 2013, Adrian et al. 2015]), and industry initiatives for software security (e.g., [Safecode 2011, OWASP , CYBSI 2014]). Table 1 shows the grouping of misuse categories, misuse main categories, and subsets.

Cryptography misuses are not all equally difficult to avoid [Braga and Dahab 2016]: some are easier to find and correct than others, depending on the involved complexity to identify and fix misuses. There are three complexity groupings for the nine misuse categories (in Table 1):

1. **Low complexity** misuses are related to coding activities and issues in APIs, and could be easily found by simple code reviews and skilled developers (supported by tools). This group includes Weak Crypto (WC), Coding Bugs (CIB), and Bad Randomness (BR).
2. **Medium complexity** misuses are related to flaws in program design affecting a few programs and may be difficult to identify due to feature distribution across programs. This group includes Improper Certificate Validation (ICV) issues, Program Design Flaws (PDF), and Public-Key Crypto (PKC) issues.
3. **High complexity** is related to flaws in system design and architecture, and requires understanding of system architecture to analyze underlying cryptosystems. This group includes Poor Key Management (PKM), IV and Nonce Management (IVM) issues, and Crypto Architecture Issues (CAI).

3.2. Selection of Communities and Posts

The original study [Braga and Dahab 2016] selected two programming communities possibly supported by experts in applied cryptography: Oracle Java Cryptography (OJC) [OJC], a forum aimed at programming with Java Cryptographic Architecture (JCA), and Google Android Developers (GAD) [GAD], a forum for Android programming.

The reasons to choose these two communities follows. Both OJC and GAD share the same Java-based API for the Java Cryptographic Architecture (JCA) [Oracle], thus limiting the knowledge required by a code reviewer to four aspects: Java programming, JCA, Android security, and applied cryptography. Also, JCA offers a stable and generic API, which has been used **for a long time** by a large number of developers for both server-side applications and mobile devices. Furthermore, JCA was adopted by the Android platform as its main API for cryptographic services. These two communities together reach a large number of ordinary developers, most

Table 1. Classification of cryptography misuse from a developer's viewpoint.

Low complexity		Medium complexity		High complexity	
Cat.	**Misuse subtype**	**Cat.**	**Misuse subtype**	**Cat.**	**Misuse subtype**
WC	-Risky/broken crypto -Proprietary crypto -Determin. symm. enc. -Risky/broken hash -Risky/broken MAC -Custom implement.	PDF	-Insec. default behavior -Insecure key handling -Streamcipher:insec. use -Insecure combo encr./auth -Insecure combo encr./hash -Side-channel attacks	IVM	-CBC w/ non-random IV -CTR with static counter -Hard-coded or const. IV -Reused nonce in encrypt.
CIB	-Wrong configs for PBE -Common coding errors -Buggy IV generation -Null cryptography -Leak/Print of keys	ICV	-No validation of certs -Broken SSL/TLS channel -Incomplete cert. valid. -Improper valid. host/user -Wildcards certs -Self-signed certs	PKM	-Short/improper key size -Hard-coded/const. keys -Hard-coded PBE passw. -Streamcipher:reused key -Use of expired keys -Key distrib. issues
BR	-Use of Statistic PRNG -Predictible seeds -Low-entropy seeds -Static, fixed seeds -Reused seeds	PKC	-Determ. encryp. RSA -Insec. padding RSA enc. -Weak configs RSA enc. -Insec. padding RSA sign. -Weak RSA sign. -Weak ECDSA sign -Key agr.: DH/ECDH -ECC: insecure curves	CAI	-Crypto agility issues -API misunderstanding -Multiple access points -Randomness issues -PKI and CA issues

of them are supposed to be non experts in cryptography. These assumptions may not hold for specialized communities with other APIs, such as openssl [OpenSSL] or bouncy castle [BC].

Collected posts comprised a time period of five years, from January 2011 to December 2015. Posts were listed by date (newest first) and manually saved as PDF files.

OJC was the most active community, with the most posts in the selected time period. GAD is very active in general, but showed less activity for cryptographic matters. For OJC, 310 posts were collected, and the 155 most viewed were selected for further analysis (50% of total). In GAD, a pre-analysis showed that specific keywords, such as "encryption", "hash" and "sign", were covered by the more general keywords "cryptography" and "encryption", which were used to select posts. For GAD, 170 posts were collected and the 100 most viewed were selected for analysis.

The manual inspection with code review was the method to analyze each single post. Posts were inspected by a cryptography expert with the skills mentioned above. Each post was inspected for occurrences of misuse. Many posts were discarded for not being related to cryptography programming, showing only discussions about threats or attacks. After discarding, OJC data set was reduced to 140 posts and GAD achieved 71 posts.

A few topics related to environment and platform specific issues were identified: configuration and installation issues, key storage and recovery issues, bug found or reported, tool misuse or misunderstanding, interoperation issues (e.g., platforms, versions, etc.), and hardware integration issues.

3.3. Selection of Developers to Evaluate

We found that most developers just ask one question to the community and never return. On the other hand, a few developers answered most questions. This fact made it possible to track users' answers and determine whether they had learned cryptography with time. These developers not only failed in giving good answers to questions related to cryptography; sometimes, they also omitted information that could prevent cryptography misuse.

In OJC, we counted 43 distinct developers who answered at least one question. Only 8 of these

developers answered 3 or more questions, corresponding to 74.5% of all answers. One developer asked 11 questions, totalling 9 evaluated developers in OJC. In GAD, we counted 97 distinct developers who answered at least one question. Only 14 of them answered 4 or more questions, corresponding to 52% of all answers. One developer asked 6 questions, totalling 15 evaluated developers in GAD. In summary, this study was conducted for two cohorts: one for a cryptography-specific forum (OJC) with 11 subjects and the other for a general-purpose forum (GAD) with 15 subjects.

The average lifespan for OJC developers was about 22 months with a standard deviation of around 9 months. The average lifespan for GAD developers was about 20 months with a standard deviation around 11 months. Table 2 shows lifespans and number of posts for both OJC and GAD. OJC developers are identified by J# and GAD developers are identified by G#.

Table 2. Selected developers, their lifespans (in months) and number of posts.

OJC Developers			GAD Developers					
OJC#	**Lifespan**	**# of posts**	**GAD#**	**Lifespan**	**# of posts**	**GAD#**	**Lifespan**	**# of posts**
J#1	8.4	7	**G#1**	14.4	6	**G#10**	8.1	6
J#2	30.5	14	**G#2**	22.5	6	**G#11**	10.9	4
J#3	26.4	33	**G#3**	25.1	6	**G#12**	17.2	4
J#4	19.9	6	**G#4**	17.1	5	**G#13**	7.3	4
J#5	27.1	13	**G#5**	36.0	8	**G#14**	7.1	5
J#6	29.6	36	**G#6**	22.3	11	**G#15**	11.9	5
J#7	31.1	11	**G#7**	28.8	13	**-**	-	-
J#8	8.3	5	**G#8**	45.4	29	**-**	-	-
J#9	16.1	3	**G#9**	23.9	11	**-**	-	-

3.4. Method for Clustering Developer's Lifespan

In online communities, interaction among users has a chronological order, but does not have to follow simultaneous events for synchronization of activities. For instance, a developer can show a very active participation for a few months and never return, while another can have a consistent participation for a couple of years.

We noticed that users participate in communities within different lifespans, diverse in length (duration) and number of posts. A user's lifespan is counted from the first to the last participation found in the period of study. In order to capture developers' distinct lifespans within the studied time period, we adopted a simple clustering technique to split the activity in each developer's lifespan (e.g., all posts for a user) into a defined number of clusters, as described next.

Clustering is a technique for combining observed objects into groups, segments, or clusters [Murthy 2015]. Its goal is to partition the observations into groups ("clusters") so that the differences among elements assigned to the same cluster tend to be smaller than among elements in different clusters [Friedman et al. 2009].

Clustering results need to be tied to specific semantic interpretations and applications [Murthy 2015]. Therefore, it is important to utilize expert knowledge to identify clusters [Murthy 2015]. We observed a natural fit between clustering methods and the life cycle presented next.

We devised a method to normalize lifespans and compare them. Our method consists in associating a life cycle to a lifespan. A life cycle is a qualitative sequence of phases. Lifespans use absolute time scales and are quantitative, while life cycles are relative and subjective, being qualitative in nature. A life cycle is divided into five phases according to the progress the user in evaluation is supposed to have had in his lifespan. The appropriate number of phases was apparent from prior knowledge about the data set. The five phases are the following:

1. an **entrant** or newbie is a new member or an inexperienced newcomer;
2. a beginner or **novice** is starting to learn crypto and taking part in communities;
3. an accustomed **fellow** is a regular user involved in the same activities of others;
4. an **expert** has knowledge or skills in cryptography gained over a period of time;
5. a **veteran** has long experience in cryptography and is considered knowledgeable.

This is an optimistic life cycle because it supposes developers improve their skills in cryptography with time. In fact, this is a strong assumption that may not hold for most developers.

The general idea of our clustering method is to apply a divisive clustering technique in order to distribute posts through five phases according to that user lifespan. The segmentation starts with all posts in one cluster (the whole lifespan) and iteratively splits existing clusters into two smaller clusters, until it satisfies a termination condition, e.g. the desired number of clusters. Roughly speaking, the lifespan is the bigger cluster, which we divide in five slices (life-cycle phases) in three iterations of the clustering method.

The distribution of posts through phases (i.e. cluster assignments, e.g. the assignment of posts to clusters) were refined by repeatedly attempting subdivision, and keeping the best resulting splits. In our method, a short lifespan lasts less than one year and a long lifespan lasts more than one year. We managed to distribute posts uniformly whenever it is possible, according to the following rules:

1. when there were less than five posts in the lifespan, we managed to put two at the ends and, optionally, two (or one) near the middle;
2. posts within the same month (a small time lapse) were put in the same phase (cluster);
3. when there was a large time lapse (bigger than 1 or 2 months in short lifespans or bigger than 4 to 6 months in long lifespans) between two consecutive posts, we adopted this gap as a phase separator (split point).

In general, divisive clustering methods, as the one we adopted, encounter difficulties regarding the selection of split points, leading to low-quality clusters [Murthy 2015]. We achieved a balanced distribution in general and a similar amount of posts into clusters, for both communities, with slightly more posts found at middle phases (Novice, Fellow, and Expert), and fewer posts found in first and last phases (Entrant and Veteran). Figure 1 shows the distribution of posts throughout the five phases for OJC and GAD.

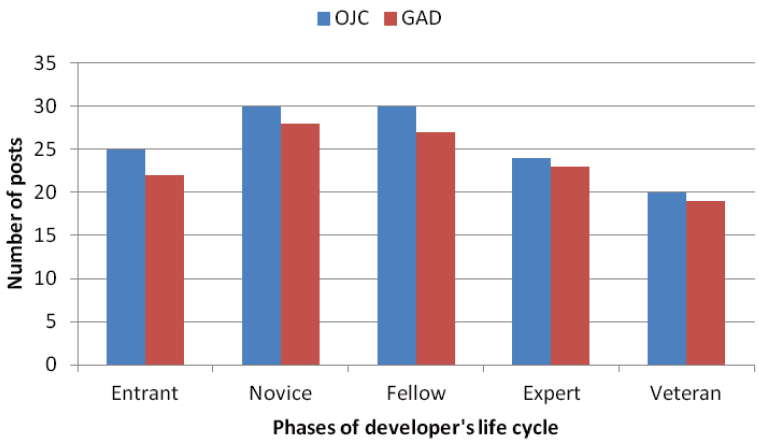


Figure 1. Distribution of posts through life cycle phases.

4. Results and Findings

This section analyses results for both communities in two distinct measures: misuse count and misuse density. Also, misuse density is analyzed in the context of the complexity groupings in Table 1. Before, we show general statistics about cryptography misuse in Table 3, that uses data from [Braga and Dahab 2016].

Table 3. Crypto misuse in online communities, from [Braga and Dahab 2016].

Categories of Cryptography Misuse	Communities	
	OJC	GAD
Weak Cryptography (WC)	26%	21%
Coding and Implementation Bugs (CIB)	17%	17%
Bad Randomness (BR)	0%	1%
Program Design Flaws (PDF)	6%	8%
Improper Certificate Validation (ICV)	4%	3%
Public-Key Cryptography (PKC) issues	16%	10%
Poor Key Management (PKM)	11%	4%
IV/Nonce Management (IVM) issues	5%	6%
Crypto Architecture Issues (CAI)	20%	1%
Platform Specific Issues (PSI)	60%	32%

The occurrence of cryptography misuse for each community is summarized in Table 3, which shows that weak cryptography (WC) is the most common misuse in both OJC and GAD. Also, both communities suffer negative influence from platform-specific issues (PSI). Besides specific issues, OJC suffers the most influence from weak cryptography (WC, 26%), architectural issues (CAI, 20%), coding bugs (CIB, 17%), public-key issues (PKC, 16%), and poor key management (PKM, 11%). These numbers are due to API misuse, lack of knowledge in applied cryptography, and complexity of JCA.

GAD suffers most from weak cryptography (WC, 21%), coding bugs (CIB, 17%), and public-key issues (PKC, 10%). These numbers are due to API misuse and lack of knowledge in cryptography programming. Despite preserving the Java API, Android has its own architecture for enabling cryptographic libraries, which simplifies installation and configuration, but brings new interoperational issues. Also, misuse of Pseudo-Random Number Generators (PRNGs) was barely mentioned in both communities, suggesting that developers have no doubts about simple uses of Java's SecureRandom API.

4.1. Cryptography Misuse Count per Life Cycle Phase

Figure 2 shows misuse counts for OJC (left) and GAD (right), for all misuse categories, distributed along the five phases of developer's life cycle. These communities have distinct behavior.

In OJC (left), developers start with a shy participation with relatively few misuses in Entrant phase. It is possible to observe that Novice is the phase with most misuses, with a notable presence of categories CAI, PKC, and WC. Also, OJC developers seem to be improving their skills in cryptography, because the total count of misuses gradually decreases from Novice to Veteran phase. However, the numbers for simple misuses (e.g., WC and CIB) are relatively stable (not decreasing), suggesting that simple misuses are recurrent and developers are not getting better at them. PKM and PKC are common issues in early phases (Novice and Fellow), with higher values, suggesting that developers are improving in these categories. Also, the number of CAI issues decreases from Novice to Veteran, suggesting that knowledge about Java's crypto API increases with time.

In GAD (right side), developers start in the Entrant phase with an expressive participation, having most misuses in WC and CIB. The notable decrease in crypto misuse for Novice and

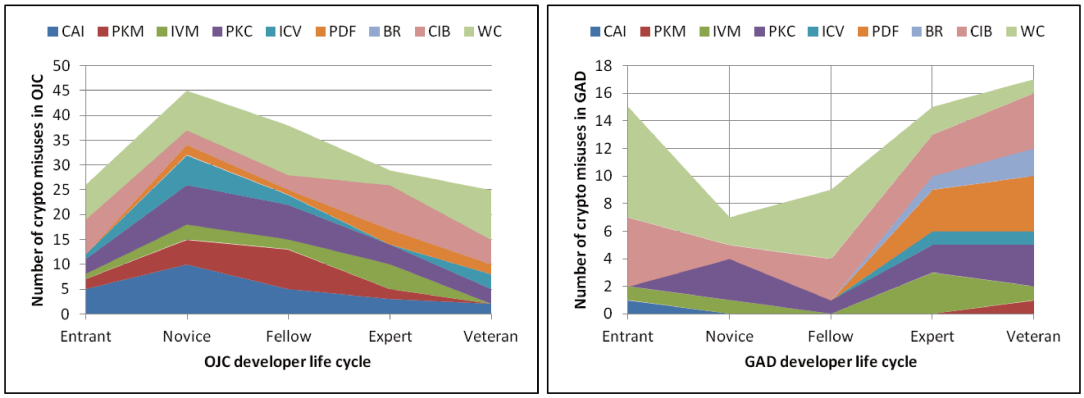


Figure 2. Crypto misuse counts for life cycle phases: OJC (left) and GAD (right).

Fellow phases suggests developers have a fast learning curve for those misuses less influenced by platform specific issues. However, in the Expert and Veteran phases the number of misuses increases again due to the influence of platform issues associated to sophisticate and complex misuses (e.g., PDF, PKC, ICV, and IVM). Four misuse categories (WC, CIB, PKC, and IVM) are recurrent in all phases, suggesting Android's diversity of hardware and software makes it difficult to learn cryptography not only for simple misuses, but also for complex misuses.

4.2. Cryptography Misuse Density per Life Cycle Phase

In order to measure crypto misuse density, we adapted the traditional metric for issue density per unit of size, which is also named defect density [Pandian 2003], bug density [Hutcheson 2003], and fault density [Bourque and Fairley 2014]. In order to measure how developers misuse cryptography, we counted the number of misuses that have been detected in posts for a developer and normalized this measure by the total number of posts for the developer in question, obtaining a value for misuse density. A straightforward analysis on misuse density over time was used to evidence a learning curve for developers as well as to show misuse reduction (or growing) as a trend.

Figure 3 shows the misuse density for OJC throughout the life cycle in two charts. On the left, crypto misuse per post (c.m.p.p) is compared to platform issues per post (p.i.p.p). In this chart, it is possible to observe that misuse density is relatively stable over time, despite a gradual reduction in density for platform issues. The chart on the right shows misuse density for low-, medium- and high-complexity misuses. In this chart, it is possible to observe that density of simple misuses (WC, CIB, and BR) increases over time, while density of moderate (PDF, PKC, and ICV) misuses has a small decrease, and density for high-complexity misuses (CAI, PKM, and IVM) shows a gradual decrease.

This behavior suggests that simple misuses are recurrent in OJC and do not depend on the actual knowledge of Java's crypto API. On the other hand, medium-complexity misuses are the most influenced by platform issues, closely following p.i.p.p behavior. Then, complex misuses (related to system design and architecture) decreases over time, suggesting a gradual improvement in developer's knowledge about Java's crypto API.

Similarly, Figure 4 shows the misuse density for GAD throughout the life cycle in two charts. On the left, it is possible to observe that misuse density increases over time (the opposite behavior of OJC), despite a relatively stable density of platform issues. The chart on the right shows that density for complex misuses is relatively stable, while density for low- and medium-complexity misuses grows over time. In fact, GAD developers start with a high density for simple misuses and,

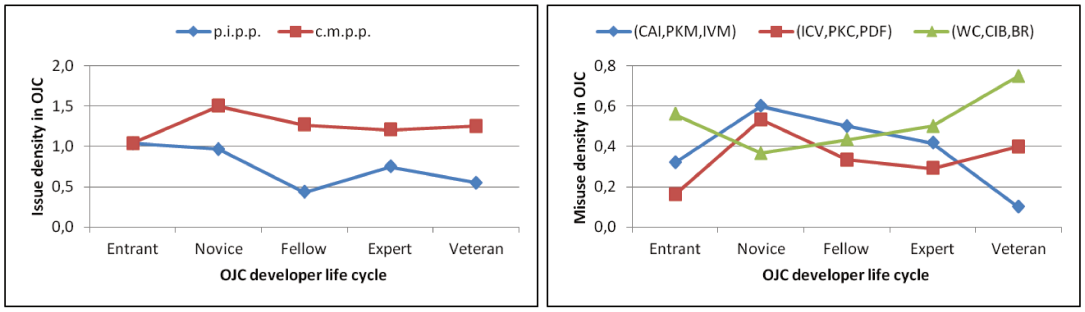


Figure 3. Misuse density in OJC compared to issue density and complexity groups.

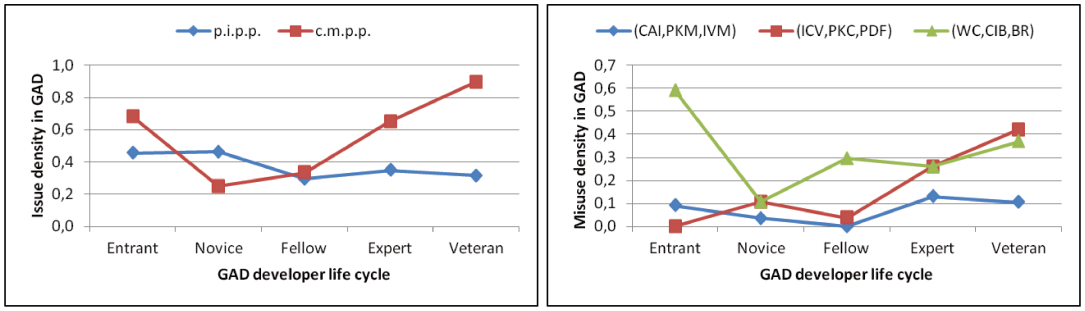


Figure 4. Misuse density in GAD compared to issue density and complexity groups.

after a sharp drop, this density gradually grows. This behavior suggests that Android developers are not getting better in cryptography over time.

5. Two Life Cycles in Detail

This section details the life cycles of two developers. The objective is to illustrate with real cases the recurrence of crypto misuse in developers' lifespans. We selected two users with more posts and longer lifespans, being good representatives of their communities: J#6 from OJC with a lifespan of 29.6 months and 36 posts, and G#8 from GAD with a lifespan of 45.4 months and 29 posts. These developers not only failed to give good answers, but sometimes omitted information that could prevent crypto misuse. Also, we could not identify improved cryptography skills for these developers.

5.1. Life Cycle for a Java Developer (J#6)

In the entrant phase, J#6 contributed to posts related to incompatible crypto providers, buggy hardware modules, cross-platform verification of certificates, and cross-language (e.g., from C++ to Java) encryption. Misuses were associated to hard-coded keys and IVs, wrong ciphertext encoding, weak cryptography with custom implementations, short keys, misconfigured RSA, and deterministic encryption with RSA.

In the novice phase, J#6 experienced misuses associated to deterministic symmetric encryption (AES/ECB), broken cryptography (DES), proprietary cryptography with custom key agreement, and coding errors. Other misuses were associated to unsafe defaults, insecure padding for RSA, deterministic encryption or short keys for RSA, and issues for DH and ECC. Complex misuses were associated to insufficient length and key distribution issues, as well as IV misconfiguration and design flaws in cryptographic architecture.

In the fellow phase, J#6 contributed to posts related to fails in TLS authentication, RSA encryption, digital signature verification, and proprietary encryption. Misuses were associated

to buggy PKI software, proprietary cryptography, risky cryptography (3DES in ECB), coding errors in insecure key handling, insecure padding or inadequate key length for RSA.

In the expert phase, J#6 still talked about coding errors when using AES with password-based encryption, hard-coded IVs and deterministic encryption, misunderstanding digital certification, deterministic signatures with RSA, and insecure key derivation. Platform issues led to misunderstand of PKI functions. Other misuses were associated to flawed IV generation, broken hash function, unsafe default, insecure padding for RSA, non-random or constant IVs, and reuse of keys with stream ciphers.

In the veteran phase, J#6 could not give correct answers to posts related to MAC with broken hash (e.g., MD5), encoding of keys, issues in key generation, and misconfigured PKI software. Other misuses were associated to public-key issues (insecure padding for RSA, and misconfigured DH), unsafe defaults, improper certificate validation (non-validated hostname and self-signed certificates), coding errors disabling cryptography, and deterministic symmetric encryption.

5.2. Life Cycle for an Android Developer (G#8)

In the entrant phase, G#8 was involved in several discussions about password-based encryption, errors when decrypting data from strings, use of SHA1 to generate keys from passwords, use of AES in CBC mode to encrypt files, and Android's full encryption. Misuses were related to broken encryption and hashes, misconfigured PBE, insecure deterministic encryption, ciphertext encoding errors, custom implementation of PBE, and constant IVs.

In the novice phase, G#8 was involved in posts related to buggy implementations of ECC in SSLv2 and signature verification on crypto libraries, proprietary implementation of SSL, and use of RSA encryption. Several misuses were associated to insecure padding and deterministic encryption for RSA, custom implementation of SSL, and attempts to use buggy implementations of ECC.

In the fellow phase, G#8 discussed cryptography adopted by Google Drive, errors when using the wrap method for protection of keys with PBE in specific versions of Android, and cross-platform verification of signatures (Java and dotNET). Misuses involved misconfigured PBE with small parameters, use of risky hashes and broken encryption, errors in ciphertext encodings, and insecure padding and deterministic encryption for RSA.

In the expert phase, G#8 was involved in discussions about several errors related to bad padding in encryption with AES, parsing keys from certificates for RSA encryption, and backward incompatibility of encryption algorithms in Android. Misuses associated to improper certificate validation, insecure defaults, deterministic encryption with RSA, non-random or constant IVs, and misconfigured PBE.

In the veteran phase, G#8 was involved in posts related to cryptography issues in Android, such as storage and recovery of keys from the device's keystore, cross-version decryption of files, and encoding ciphertext as integers. Misuses were associated to improper certificate validation with self-signed root certificates, misconfigured PBE with small parameters, insecure defaults for AES, non-random IVs, and ciphertext encoding errors.

6. Discussion

We are aware that our analysis have to be put in context and is restricted to the main subject of the two communities evaluated. That said, we tried to generalize our conclusions.

For developers, as much as for end-users, security is a secondary concern. Developers usually have priorities (e.g., functional correctness, time to market, maintainability, economics, compliance

with other corporate policies) that often appear to conflict with security. Frequently, developers look for quick, but insecure solutions and online communities favor this behavior.

Ideally, developers should not be forced to learn cryptography in order to correctly use cryptographic APIs, specially for simple use cases. However, in practice, crypto APIs are unable to foster their correct use without domain knowledge obtained from elsewhere but online communities.

Java has a stable API, with a very predictable behavior, favoring developers with enough time to understand its particularities. In general, developers improve their skills in misuse categories affected by platform issues, but this does not happen for simple misuses. On the other hand, Android uses the same crypto API of the Java platform, but this fact alone is not enough to promote a positive learning curve for cryptography. Many issues related to diversity of both hardware and software negatively affect how developers learn cryptography in Android.

Java cryptographic architecture presents issues with installation and configuration that divert developers from actual tasks of cryptographic programming. Also, specific issues showed up when integrating Java programs to cryptographic hardware or communicating with applications in other platforms. These troublesome issues frequently obfuscate crypto misuses in the same code. For instance, in Java, a worst-case scenario occurs when developers write buggy code for encrypting data and use weak cryptography by accidentally adopting insecure defaults.

Android solved many issues faced by Java developers, but brought to daily troubleshooting several interoperation issues due to the diversity of both hardware and software in that platform. Developers, confused by these issues, are distracted from actual cryptographic pitfalls. For instance, developers suffering from interoperation issues among devices used weak cryptography to protect stored passwords, and derived keys directly from password hashes.

Developers learn how to make APIs work, but this does not mean cryptography was used correctly. In fact, coding bugs are persistent issues when using general-purpose (function-based) crypto APIs to implement application-specific use cases, because developers are forced to make insecure choices without actually understanding the whole situation. This suggests developers would benefit from high-level cryptographic frameworks (oriented toward use cases) or task-based APIs that could avoid simple misuses and insecure design decisions.

The overabundance of complex options for security leads to disengagement when confronted by other concerns. We have noticed that complex architectures distract developers from actual cryptographic misuse and contribute to perpetuate issues in cryptographic programming. For instance, one curious reason developers gave to use homemade code for cryptographic algorithms is to avoid dependencies to external libraries.

Finally, we did not see in developers' lifespans any posts concerned with the insecure combination of encryption and authentication, padding-oracle attacks, or selection of insecure elliptic curves. This suggests that these developers never learned about design flaws for authenticated encryption, side-channel attacks or obsolete implementations for elliptic curve cryptography. Another remarkable absence in lifespans was the concern with weak parameters in public-key cryptography (e.g., RSA, DH), suggesting that developers always take for granted the quality of parameters generated by tools.

7. Concluding Remarks

Ordinary software developers are used to obtain quick solutions to daily problems from fellows in online communities. Those communities associated to cryptographic programming are good for finding solutions to platform-specific issues (e.g., implementation bugs, incompatible hardware, and misconfigured software) as well as to clarify obscured aspects of API usage. On the other

hand, developers willing to learn cryptography from online communities, if lucky, receive only shallow advice and usually do not have a positive learning curve over time. Our study shows that cryptography misuse is perpetuated in online communities and frequently reappear in recurrent issues, because these communities favor quick, but insecure solutions and even active developers (of those communities) take security as a secondary concern.

We believe this longitudinal study effectively contributes to better understanding how cryptography is handled by ordinary developers of two online communities, bringing to light their attitudes and priorities concerning cryptography-based security over time. It is paramount to improve APIs to increase usability and to foster best practices. Also, there are opportunities for future work in behavioral experiments of cryptographic programming, surveys with actual developers, as well as replication studies focusing on other communities and crypto APIs.

Acknowledgements

We thank Intel and CNPq for the financial support, as well as CPqD and UNICAMP for the institutional support.

References

Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M. L., and Stransky, C. (2016a). You Get Where You're Looking For: The Impact Of Information Sources on Code Security. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 289–305. IEEE.

Acar, Y., Fahl, S., and Mazurek, M. L. (2016b). You Are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users.

Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., Heninger, N., Springall, D., Thomé, E., Valenta, L., and Others (2015). Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17. ACM.

Alashwali, E. S. (2013). Cryptographic vulnerabilities in real-life web servers. In *Third International Conference on Communications and Information Technology (ICCIT)*, pages 6–11. Ieee.

BC. The Legion of the Bouncy Castle.

Bos, J. W., Halderman, J. A., Heninger, N., Moore, J., Naehrig, M., and Wustrow, E. (2014). Elliptic curve cryptography in practice. In *Financial Cryptography and Data Security*, pages 157–175. Springer.

Bourque, P. and Fairley, R., editors (2014). *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society, version 3. edition.

Braga, A. and Dahab, R. (2015). Introdução à Criptografia para Programadores: Evitando Maus Usos da Criptografia em Sistemas de Software. In *Caderno de minicursos do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2015*, pages 1–50.

Braga, A. and Dahab, R. (2016). Mining Cryptography Misuse in Online Forums. In *2nd International Workshop on Human and Social Aspect of Software Quality*.

Chatzikonstantinou, A., Ntantogian, C., Xenakis, C., and Karopoulos, G. (2015). Evaluation of Cryptography Usage in Android Applications. *9th EAI International Conference on Bio-inspired Information and Communications Technologies*.

Chess, B. and West, J. (2007). *Secure programming with static analysis*.

CYBSI (2014). Avoiding The Top 10 Software Security Design Flaws.

Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. (2013). An empirical study of cryptographic misuse in android applications. *ACM SIGSAC conference on Computer & comm. security (CCS'13)*, pages 73–84.

Fahl, S., Harbach, M., and Muders, T. (2012). Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *ACM conference on Computer and communications security*, pages 50–61.

Friedman, J., Hastie, T., and Tibshirani, R. (2009). *The elements of statistical learning*, volume 2. Springer-Verlag.

GAD. Google Android Developers.

Georgiev, M., Iyengar, S., and Jana, S. (2012). The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*, pages 38–49.

Howard, M. and LeBlanc, D. (2003). *Writing secure code*.

Howard, M., LeBlanc, D., and Viega, J. (2009). *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill Education.

Howard, M. and Lipner, S. (2006). *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA.

Hutcheson, M. L. (2003). *Software testing fundamentals: methods and metrics*. John Wiley & Sons.

Lazar, D., Chen, H., Wang, X., and Zeldovich, N. (2014). Why Does Cryptographic Software Fail?: A Case Study and Open Problems. In *5th Asia-Pacific Workshop on Systems*, APSys '14, pages 7:1—-7:7, New York, NY, USA. ACM.

Mart, V. G. and Hern, L. (2013). Implementing ECC with Java Standard Edition 7. *International Journal of Computer Science and Artificial Intelligence*, 3(4):134–142.

Murthy, M. R. (2015). *Introduction to Data Mining and Soft Computing Techniques*. Laxmi Publications.

Nadi, S., Krüger, S., Mezini, M., and Bodden, E. (2016). "Jumping Through Hoops": Why do Java Developers Struggle With Cryptography APIs? *The 38th International Conference on Software Engineering*.

OJC. Oracle Java Cryptography.

OpenSSL. OpenSSL Cryptography and SSL/TLS toolkit.

Oracle. Java Cryptography Architecture (JCA) Reference Guide.

OWASP. Cryptographic Storage Cheat Sheet.

Pandian, C. R. (2003). *Software metrics: A guide to planning, analysis, and application*. CRC Press.

RWC. Real World Crypto Symposium.

Safecode (2011). Fundamental Practices for Secure Software Development.

Shostack, A. (2014). *Threat modeling: Designing for security*. John Wiley & Sons.

Shuai, S., Guowei, D., Tao, G., Tianchang, Y., and Chenjie, S. (2014). Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In *IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC)*, pages 75–80.

Viega, J. and McGraw, G. (2001). *Building Secure Software: How to Avoid Security Problems the Right Way*.

Wang, W. and Godfrey, M. W. (2013). Detecting API Usage Obstacles : A Study of iOS and Android Developer Questions. pages 61–64.

Wang, W., Malik, H., and Godfrey, M. W. (2015). Recommending posts concerning api issues in developer q&a sites. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 224–234. IEEE Press.

2.2 Evaluation of automated tools for cryptography

This section contains the following publications. First, the publication entitled *"A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software"*. Second, the publication entitled *"Practical Evaluation of Static Analysis Tools for Cryptography: Benchmarking Method and Case Study"*.

2.2.1 A Survey on Tools and Techniques for Programming and Verification of Secure Cryptographic Software

This publication is entitled *"A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software"* and was published at the XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg 2015), held in the city of Florianópolis, Brazil.

A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software

Alexandre Braga[12], Ricardo Dahab[2]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Av. Albert Einstein, 1251 – 13083-852 – Campinas – SP – Brazil

[2]Centro de Pesquisa e Desenvolvimento em Telecomunicações (Fundação CPqD)
R. Dr. Ricardo Benetton Martins, S/n – 13086-902 – Campinas – SP – Brazil

ambraga@cpqd.com.br, rdahab@ic.unicamp.br

Abstract. *This paper contributes to broaden the discussion on tools and techniques in cryptographic programming and verification. The paper accomplishes three goals: (i) surveys recent advances in supporting tools for cryptographic software programming and verification; (ii) associates these tools to current security practices; and (iii) organizes their use into software programming and verification steps. The paper concludes that there is no single tool for secure development of cryptographic software. Instead, only a well-crafted toolkit can cover the whole landscape of secure cryptographic software coding and verification.*

1. Introduction

Today's software systems exist in a world full of massively available, cloud-based applications (e.g., mobile apps) that are always on-line, connected to whatever servers are available, and communicating to each other. In this world, as more private aspects of life are being carried out through mobile devices, apps act on behalf of their users as proxies of their identities in everyday activities, processing, storing and transmitting private or sensitive information. An increasing number of threats to this information make computer security a major concern for government agencies, service providers, and even ordinary people, the consumers of modern gadgets and digital services.

In these circumstances, it is quite natural to observe a rise in the use of security functions based on cryptographic techniques in software systems. Moreover, the scale of encryption in use today has increased too, not only in terms of volume of encrypted data (the newest smartphones have encrypted file systems by default), but also relating to the amount of applications with cryptographic services incorporated within their functioning (for instance, an app store has hundreds of apps advertising cryptographic protections, see https://play.google.com/store/search?q=cryptography&c=apps). In addition to the traditional use cases historically associated to stand-alone cryptography (e.g., encryption/decryption and signing/verification), there are several new usages strongly related to final user's needs, transparently blended into software functionalities, and bringing diversity to the otherwise known threats to cryptographic software.

In spite of Cryptography's popularity, however, we believe that the software industry lacks an approach for building secure cryptographic software in common software factories, which could be used by ordinary programmers. This paper contributes to remedy this situation, broadening the discussion on the development of secure cryptographic software by addressing the use of security tools and techniques during the various stages of cryptographic programming. It also contributes to widen the scope of single tools, by providing a more comprehensive view of the landscape of tools and techniques in cryptographic programming. The paper accomplishes its objectives in three ways: (i) surveying recent advances in supporting tools for cryptography programming; (ii) associating these tools to current security practices whenever possible; and (iii) organizing their use into the current stages of software programming and verification.

The text is organized as follows. Section 2 offers background and motivation. Section 3 surveys tools and techniques for cryptographic programming and verification. Section 4 organizes the surveyed tools and techniques into the stages of software development. Section 5 concludes this paper and discusses future work.

2. Background and motivation

In this text, cryptographic software is one that has as its very purpose a true need for securing or preserving some of the information security's main goals (namely integrity, authenticity, confidentiality, and non-repudiation) through the use of cryptographic technology. To accomplish these goals, the software can use cryptography directly, by means of proprietary implementations, or through reusable libraries and frameworks. Either way, implementations of cryptographic algorithms must be carefully constructed to be free of problems that compromise software security. Also, these secure implementations have to be securely used by application programmers, who take for granted the quality of the algorithms' internals.

In spite of the four decades since the golden rules of software security were published by Saltzer and Schroeder [1], secure software engineering [2][3] seems not to directly address the issue of cryptographic security. Actually, for almost twenty years, studies have shown that vulnerabilities in cryptographic software have been mainly caused by software defects and poor management of cryptographic parameters and other sensitive material [4][5][6][7][8].

Furthermore, recent studies [9][10][11][12][13][14][15] showed the recurring occurrence of well-known bad practices of cryptography usage in various software systems (mobile apps in particular). In fact, since 2012, the interest of academia in "Real World Cryptography" [16] has become explicit and is gaining momentum.

It is important to differentiate secure (or defensive) programming of cryptographic software from programming secure cryptographic software. The former is related to the use of general secure coding techniques during the programming of cryptographic software. The latter starts at the point where the former stops, and embraces specific secure coding techniques and programming countermeasures to better defend cryptographic software against particular misuses as well as bad construction of cryptographic techniques.

Today, cryptographic software of recognized quality is generally treated as a "work of art", whose constructive process can hardly be reproduced by the average programmer. For the diligent observer, there is a proliferation of bad implementations as well as lack of good ones. A number of modern examples support this statement:

- The inadequacy of current software tools (e.g., SDKs, programming languages, and compilers, etc.) to cope with security issues in cryptographic programming. For instance, see the recent occurrences of well-known vulnerabilities in cryptographic software (e.g., HeartBleed [17] and Apple's GoTo Fail bug [18]).
- Low-level cryptographic services have been misused by ordinary programmers without proper instrumentation or education. Basic cryptographic processes are not being effectively learned by programmers [10][11][14][15], who make the same mistakes over and over.
- Sophisticated security concepts are not being well presented by existing frameworks. For instance, validation of digital certificates forces unnecessary complexity onto programmers due to misunderstanding of how cryptographic frameworks should be shaped [9][12][13].
- Vulnerabilities ultimately related to architectural aspects of cryptographic services and API design can expose unexpected side-channels and leak information. For instance, Padding Oracles [19] can occur due to inappropriate error handling at upper layers when orchestrating cryptographic services and potentially leak information [20][21][22].

2.1. Recent studies on cryptographic bad practices

This section analyzes recent studies on misuse commonly found on cryptographic software. According to recent studies by Egele et al [11] and Shuai et al [14], the most common misuse is the use of deterministic encryption, where a symmetric cipher in Electronic Code Book (ECB) mode appears mainly in two circumstances: AES/ECB and 3DES/ECB. There are cases of cryptographic libraries in which ECB mode is the default option, automatically selected when the operation mode is not explicitly specified by the programmer. A possibly worse variation of this misuse is the RSA in Cipher Block Chaining (CBC) mode without randomization, which is also available in modern libraries, despite of being identified more than 10 years ago by Gutmann [5].

Another frequent misuse is hardcoded initialization vectors (IVs), even with fixed or constant values [11]. Initialization vectors, in almost all operation modes of block ciphers, must be both unique and unpredictable. The exception is the CTR mode, which requires unique IVs (without repetition). This requirement is extended to the authenticated encryption mode GCM. A related misuse is the use by the ordinary programmer of hardcoded seeds for PRNGs [11]. A common misunderstanding concerning the correct use of IVs arises when (for whatever reason) programmers need to change operation modes of block ciphers. For instance, the Java Cryptographic API [23] allows operation modes to be easily changed, without considering IV requirements.

The validation of digital certificates in web browsers is relatively well built and reliable, despite the fact that users usually ignore the warnings referring to invalid

certificates. In software other than web browsers, especially in mobile apps, there is a wide range of libraries for handling SSL/TLS connections. Recent studies by Georgiev et al [12] and Fahl et al [13][24] show that all of these allow the programmer to ignore some step in certificate verification in order to further usability or performance, but introducing vulnerabilities. Especially, a failure in signature verification or in domain-name verification favors the Man-in-the-Middle (MITM) attack.

Finally, a recent study by Lazar et al [15] show, from an analysis of 269 cryptography-related vulnerabilities, that just 17% of the bugs are in cryptographic libraries, and the remaining 83% are misuses of cryptographic libraries by applications.

3. Tools and techniques for secure cryptographic software

This section surveys tools and techniques for assisted programming and verification of cryptographic software. The set of tools and techniques were divided in two categories: secure cryptographic programming and security verification of cryptographic software.

3.1. Secure cryptographic programming

Secure programming tools of cryptographic software consist of specific programming languages, tools for automated code generation, cryptographic APIs, and frameworks.

3.1.1. Cryptographic programming languages

The use of specific programming languages is not standard practice in secure software development. On the other hand, experts, such as cryptologists, usually prefer their knowledge expressed in its own syntax by domain-specific languages [25][26][27][28].

cPLC [25] is a cryptographic Programming Language and Compiler for generating Java implementations of two-party cryptographic protocols, such as Diffie-Hellman. cPCL's input language is strongly inspired by the standard notation for specifying protocols and is, allegedly, the first tool which can be used by cryptographically untrained software engineers to obtain sound implementations of arbitrary two-party protocols, as well as by cryptographers who want to efficiently implement their protocols designed on paper.

Barbosa, Moss, and Page [26] have worked with the CAO programming language to provide a cryptography-aware domain-specific language and associated compiler intended to work as a mechanism for transferring and automating the expert knowledge of cryptographers into a form which is accessible to anyone writing security-conscious software. CAO allowed the description of software for Elliptic Curve Cryptography (ECC) in a manner close to the original mathematics, and its compiler allowed automatic production of executable code competitive with hand-optimized implementations. Recently, CAO's typing system (the set of rules to assign types to variables, expressions, functions, and other constructs on a programming language) was formally specified, validated and implemented in a way to support the implementation of front-ends for CAO compilation and formal verification tools [29]. Finally, a compiler for CAO was released [30]. The tool takes high-level cryptographic algorithm specifications and translates them into C implementations through a series of security-aware transformations and optimizations.

Cryptol [27] is a functional domain-specific language for specifying cryptographic algorithms. The language works in such a way that the implementation of an algorithm resembles its mathematical specification. Cryptol can produce C code, but its main purpose is to support the production of formally verified hardware implementations. Cryptol is supported by a toolset for formally specifying, implementing, and verifying cryptographic algorithms [31].

The last work worth mentioning is a domain-specific language for computing on encrypted data [28], which can be called Embedded Domain-Specific Language for Secure Cloud Computing (EDSLSCC). The language was designed for secure cloud computing, and it is supposed to allow programmers to develop code that runs on any secure execution platform supporting the operations used in the source code.

3.1.2. Automated code generation

Automated code generation is not a common practice in secure coding. In spite of that, it has been successfully used to generate cryptographic-aware source code. A recent work of Almeida et al [32] extend the CompCert certified compiler with a mechanism for reasoning about programs relying on trusted libraries, as well as translation validation based on CompCert's annotation mechanism. These mechanisms, along with a trusted library for arithmetic operations and instantiations of idealized operations, proved to be enough to preserve both correctness and security properties of a source code in C when translated down to its compiled assembly executable.

Another category of tools transforms code by inserting secure controls. Two recent works by Moss et al [33][34] describe the automatic insertion of Differential Power Analysis (DPA) countermeasures based on masking techniques. Another work by Agosta et al [35] performs security-oriented data-flow analysis and comprises a compiler-based tool to automatically instantiate the essential set of masking countermeasures.

3.1.3. Advanced cryptographic APIs

An application programming interface (API) is the set of signatures that are exported and available to users of a library or framework [36]. This section shows cryptographic libraries that go beyond the ordinary cryptographic API by either presenting differentiated software architectures or offering services in distinguished ways, resembling software frameworks.

Two recent studies by Braga and Nascimento [37] and González et al [38] have shown that, in spite of the observed diversity of cryptographic libraries in academic literature, these libraries are not necessarily publicly available or ready for integration with third party software. In spite of many claims of generality, almost all of them were constructed with a narrow scope in mind and prioritize academic interests for non-standard cryptography. Furthermore, portability to modern platforms, such as Android, used to be a commonly neglected concern in cryptographic libraries [37]. So much so, that modern platforms only offer by default to the ordinary programmer a few options of common cryptographic services [38].

Gutmann's Cryptlib [39] is both a cryptographic library and an API that emphasizes the design of the internal security architecture for cryptographic services, which are wrapped around an object-oriented API, providing a layered design with full isolation of architecture internals from external code. Nevertheless, Cryptlib still looks like a general-purpose cryptographic API (although an object-oriented one), presenting a collection of services encapsulated by objects. Also, it does not remove the need for cryptographic expertise, especially when composing services. The problems faced when using Cryptlib in real-world applications have already been documented [5].

NaCl, proposed by Bernstein, Lange, and Schwabe [40], stands for "Networking and Cryptography Library." NaCl offers, as single operations, compositions of cryptographic services that used to be accomplished by several steps. For instance, with NaCl, authenticated encryption is a single operation. The function *c* = crypto_box*(m, n, pk, sk)*, where *sk* is sender's private key, *pk* is the receiver's public key, *m* is the message and *n* is a nonce, encapsulates the whole scenario of public-key authenticated encryption from the sender's point of view. The output is authenticated and encrypted using keys from the sender and the receiver.

The crypto_box function has its advantages. With most cryptographic libraries, writing insecure programs is easier than writing programs that include proper authentication, because adding authentication signatures to encrypted data cannot be accomplished without extra programming work. Unfortunately, NaCl does not offer a solution to one of the most common sources of errors in the use of cryptography by ordinary programmers, namely the generation and management of nonces. NaCl leaves nonce generation and management to the function caller, under the argument that nonces are integrated into high-level protocols in different ways [40]. This issue was addressed in part by libadacrypt [41], a misuse-resistant cryptographic API for the Ada language.

3.1.4. Cryptographic frameworks

According to Fayad and Schmidt [42], an application framework is a reusable, "semi-complete'' software application that can be specialized to produce custom applications. Johnson [43] argues that, in contrast to class libraries, frameworks are targeted for particular application domains because a framework is a reusable design of a system, or a skeleton that can be customized by an application developer, providing reusable use cases. Application frameworks for cryptography have the potential for reducing coding effort and errors for complex use cases, such as certification validation, IV management, and authenticated encryption.

Nowadays there are many SSL libraries that aim at making the integration of SSL into applications easier. However, as recent studies by Georgiev et al [12] and Fahl et al [13] have shown, many of these libraries are either broken or error-prone, so that incorrect SSL validation is considered a widespread problem, and mere simplifying SSL libraries or educating developers in SSL security has not been favored as a solution to the problem [24]. Instead, the ideal solution would be to enable developers to use SSL correctly without coding effort, thus preventing the breaching of SSL validation through insecure customizations.

A recent work by Fahl et al [24] proposes a paradigm-shift in SSL usage: instead of letting developers implement their own SSL linking code, the main SSL usage patterns should be provided by operating systems' services that can be added to apps via configuration, instead of implementation. Following the intent of this new paradigm, a proposed SSL framework automates the steps in certificate validation as well as makes changes to the way SSL is currently used by mobile apps [24]. It substitutes, by configuration options, the need for writing SSL code in almost all use cases, inhibiting dangerous customizations. Also, during software development, it makes a distinction between developer devices and end-user devices, allowing self-signed certificates. Finally, problems with SSL that could result in MITM attacks are reliably informed by unavoidable OS warnings.

The idea of providing cryptography-related functions as operating system services is not new and has been already proposed for performance reasons [44][45]. The innovation in the SSL framework resides in the usability aspect, from the programmer's point of view, of having high-level SSL functionality, instead of primitive cryptography functions, as configurable OS services.

3.2. Cryptographic security verification

Security verification tools include static and dynamic analysis (testing) tools. Static analysis tools perform syntactical and semantic analysis on source code without executing it. On the other hand, testing (dynamic analysis) tools perform dynamic verifications of a program's expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain.

3.2.1. Static analysis tools

The use of automatic tools for static analysis of code is quite common in secure programming and can be considered a standard best practice. It seems quite natural that the practices of secure coding were ported to cryptographic programming as well. In fact, most coding standards [46][47] do contain simple rules concerning cryptography usage that can be easily automated by ordinary static analysis tools. For example, such tools can warn when deprecated cryptographic functions are used: uses of MD5, SHA-1, and DES can be automatically detected as bad coding, as well as the use of short keys (e.g., 128-bit key for AES or 512-bit key for RSA).

Unfortunately, there are cryptographic issues that cannot be detected by ordinary tools and simple techniques [48]. These issues have been addressed by advanced tools in academic research [11][14][49][50][51][52]. The CryptoLint [11] tool takes a raw Android binary, disassembles it, and checks for typical cryptographic misuses. The Cryptography Misuse Analyzer (CMA) [14] is an analysis tool for Android apps that identifies a pre-defined set of cryptographic misuse vulnerabilities from API calls.

The Side Channel Finder (SCF) [49] is a static analysis tool for detection of timing channels in Java implementations of cryptographic algorithms. These side-channels are often caused by branching of control flow, with branching conditions depending on the attacked secrets. The Sleuth [50] tool is an automatic verification tool attached to the LLVM compiler, which can automatically detect several examples of

classic pitfalls in the implementation of DPA countermeasures. CacheAudit [51] is an analysis tool for automatic detection of cache-side channels. It takes as input a binary code and a cache configuration and derives security guarantees based on observing cache states, traces of hits and misses and execution times. CAOVerif [52] is a static analysis tool for CAO [30] and has been used to verify NaCl code [40].

It is important to differentiate the scope of these tools. While CAOVerif, SCF, Sleuth, and CacheAudit work inside a cryptographic implementation, looking for specific types of side channels and other issues, CryptoLint and CMA work outside the bounds of cryptographic APIs and identify known misuse of cryptographic libraries.

3.2.2. Functional security tests

In cryptography validation [53], test vectors are test cases for cryptographic security functions and have been used for years in validation of cryptographic implementations, mostly for product certification post construction. According to Braga and Schwab [54], test vectors can also be used for validation during software development and accomplished by automated acceptance tests. Acceptance tests verify whether a system satisfies its acceptance criteria by checking its behaviors against requirements [36].

Test vectors are good acceptance tests because they stand halfway between cryptologists and developers [54]. Test vectors are test cases provided by cryptologists and are substitutes for requirement specifications as well as independent checks that the cryptographic software has implemented the requirements correctly. Based upon test vectors, developers can write unit tests prior to the writing of the cryptographic code to be tested. Then, test vectors can be used to evaluate the correctness of implementations, not their security. Functional correctness is a condition for security, since incorrect implementations are unreliable and insecure.

There are publicly available test vectors (e.g., [53]) which are constructed using statistical sampling. The successful validation with a statistical sample only implies strong evidence but not absolute certainty of correctness. To be statistically relevant, even small data sets possess thousands of samples, so automation is required. Once automated tests are available for cryptographic implementations, further improvements on the code can take place in order to address industry concerns like performance optimizations, power consumption and protections against side-channel attacks and other vulnerabilities. Even after all those transformations, acceptance tests preserve trust by giving strong, albeit informal, evidence of correctness [54].

3.2.3. Security testing tools

Security testing focuses on the verification that the software is protected from external attacks [36]. Usually, security testing includes verification against misuse and abuse of the software or system [36]. Security tests are as diverse as the number of exploitable vulnerabilities, and can be considered a common best practice in secure software development. It is worth mentioning the industry's current practice on cryptography testing for web applications [55][56][57] and cryptographic modules [58]. This section presents three test types and the corresponding tools, which are believed to be good representatives of current trends on security tests for cryptographic software.

In web application security, automated tests for SSL connections have been used for detection of HTTPS misconfiguration [56]. Recently, this type of test has moved to mobile applications with MalloDroid [13], a tool to detect potential vulnerabilities against MITM attack of SSL implementations on Android applications. MalloDroid performs static code analysis over compiled Android applications in order to fulfill three security goals: extract valid HTTP(S) URLs from decompiled apps by analyzing calls to networking API; check the validity of the SSL certificates of all extracted HTTPS hosts; and identify apps that contain abnormal SSL usage (e.g., contain non-default trust managers, SSL socket factories, or permissive hostname verifiers).

Padding Oracle Exploitation Tool [20] (POET) is a tool that finds and exploits padding oracles automatically. Tests against padding oracle attacks are usually hard to be performed manually due to the large number of iterations (ranging from many hundreds to a few thousand) performed to decrypt a single block of ciphertext [19]. Once a padding oracle is discovered, its exploitation can be automated by well-documented algorithms [22]. POET has been successfully used to exploit padding oracles in web technologies [20] (e.g., XML encryption [21] and ASP.NET [22]).

Finally, fault injection is a kind of fuzz testing that can be used for security testing of cryptographic devices [59]. Fuzz testing [36] is a special form of random testing (where test cases are generated at random) aimed at breaking the software. A fault-injection attack, automated by a Fault Injection Attack Tool (FIAT) is a type of side-channel attack realized through the injection of deliberate (malicious) faults into a cryptographic device and the observation of the corresponding erroneous outputs. Fault injection attacks have been shown [59] to require inexpensive equipment and a short amount of time, when compared to tests against side-channels of power consumption or electromagnetic emanations, which usually require an expensive setup.

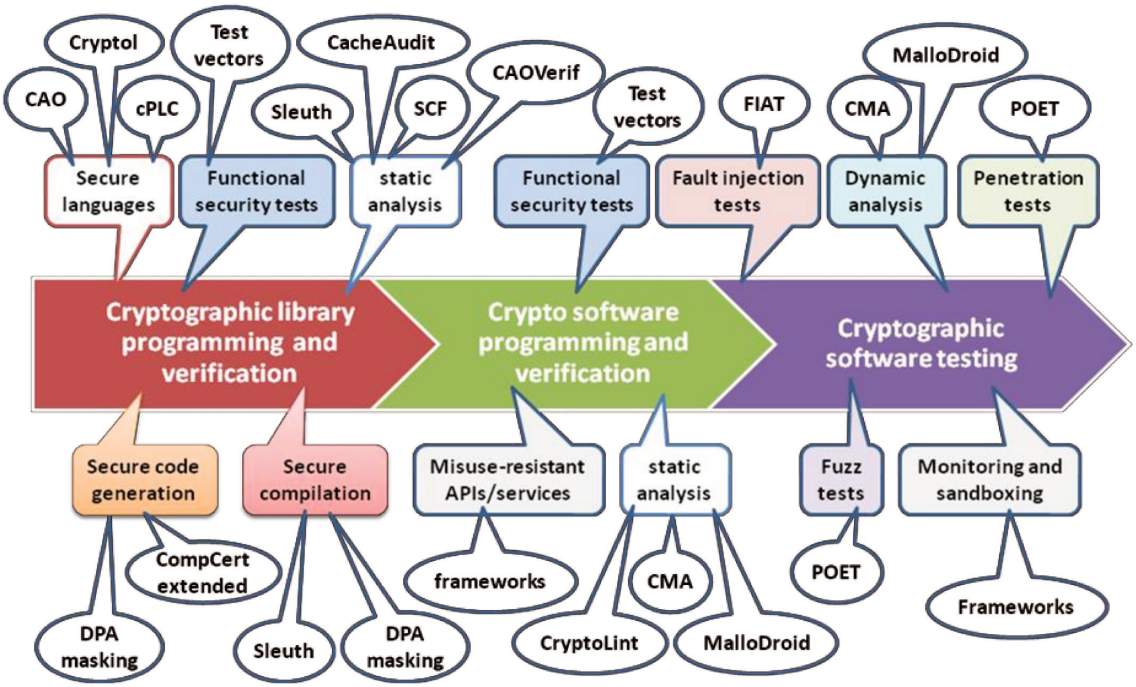


Figure 1: Automated tools in cryptographic software programming and testing.

4. Assembling a toolkit for programming cryptographic software

According to McGraw [3], there is no absolutely secure software, in the sense that the confidence in the so-called secure software is always relative to the assurance methods in use, and has to be justified by the structured use of appropriate tools and techniques.

That way, the tools and techniques discussed so far can be put together in a simple sequence of steps for programming of secure cryptographic software. Such a sequence of steps is illustrated in Figure 1 and, if put in practice, would be able to increase the confidence in the resulting cryptographic software by providing strong evidence of security. It is interesting to observe that none of the tools and techniques can alone satisfy the whole process, which can only be covered by jointly use of several tools and techniques.

The sequence has three main steps that can be performed iteratively: library programming and verification, cryptographic software programming and verification, and cryptographic software testing. In each step, appropriate tools and techniques can be placed to accomplish the required assurance. Although many tools and techniques can be used in more than one step, the major benefit can be obtained when they are used in specific (preferred) places. For instance, secure languages, secure compilation, and secure code generation are appropriate for programming of cryptographic libraries. On the other hand, static analysis tools, (automated) functional tests, and frameworks are suitable for cryptographic software programming. Finally, dynamic analysis, fault injection, (SSL) penetration tests, padding oracles tests, and monitoring are better suited for verifications.

5. Concluding remarks

Traditionally, cryptography has been considered by software developers one of the most difficult to understand security controls. Programmers were used to rely on simple APIs to grant the effectiveness of cryptography over security-related functionality, while the correctness of its internals was always taken for granted. However, the increasing complexity of software has negatively affected cryptography, leading to its misuse by programmers and ultimately resulting in insecure software.

It is now time to come up with new ways of building modern cryptographic software, by looking for benefits from recent advances on tool support for software security. The main conclusion of this paper is that there is no ultimate tool for programming secure cryptographic software. Instead, only a well-crafted set of tools seems to be able to cover the whole landscape of cryptographic software programming.

The programming of secure cryptographic software, the way it is proposed in this text, is an emerging discipline in the practice of cryptographic software programming. At the time of writing, only a few tools and techniques were actually available to the ordinary programmer. Most of tools are prototypes of academic interest and cannot be put to compete with commercial, off-the-shelf security tools. On the other hand, the software industry has a successful history of innovation in bringing new quality assurance technologies to the average programmer. So there is hope that in the

near future, a new generation of tools for software security could bring to daily practice all the academic advances mentioned so far.

The current research points to several opportunities for future work. The modeling of processes for the development of secure cryptographic software could help to find better ways of using all these tools. Also, performing experimentation with specific security tools during development of cryptographic software could provide a means to evaluate their effectiveness in reducing cryptographic vulnerabilities. The development of better software abstractions to facilitate the use of advanced cryptographic concepts by ordinary programmers is another alternative course of research. Finally, the collection and analysis of data concerning the real programming habits of developers responsible for coding cryptographic primitives could encourage further studies to better understand how programmers misuse cryptography.

Acknowledgements. Alexandre Braga would like to thank Fundação CPqD for the institutional support given to employees on their academic activities. Ricardo Dahab thanks FAPESP, CNPq and CAPES for partially supporting this work. He also thanks the University of Waterloo, where he is on a leave of absence from UNICAMP.

6. References

[1] J. Saltzer and M. Schroeder, “The protection of information in computer systems,” *Proc. of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[2] R. Anderson, “Security engineering,” 2001.

[3] G. McGraw, *Software Security: Building Security in*. 2006.

[4] B. Schneier, “Cryptographic design vulnerabilities,” *Comp.*, Sep., pp.29–33, 1998.

[5] P. Gutmann, “Lessons Learned in Implementing and Deploying Crypto Software,” *Usenix Security Symposium*, 2002.

[6] R. Anderson, “Why Cryptosystems Fail,” in *Proc. of the 1st ACM Conf. on Computer and Comm. Security*, 1993, pp. 215–227.

[7] B. Schneier, “Designing Encryption Algorithms for Real People,” *Proc. of the 1994 workshop on New security paradigms.*, pp. 98–101, 1994.

[8] A. Shamir and N. Van Someren, “Playing ‘hide and seek’with stored keys,” *Financial cryptography*, pp. 1–9, 1999.

[9] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer, “Here’s My Cert, So Trust Me, Maybe?: Understanding TLS Errors on the Web,” in *Proc. of the 22Nd Intn’l Conf. on World Wide Web*, 2013, pp. 59–70.

[10] E. S. Alashwali, “Cryptographic vulnerabilities in real-life web servers,” *2013 Third Intn’l Conf. on Comm. and Inform. Technology (ICCIT)*, pp. 6–11, 2013.

[11] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, “An empirical study of cryptographic misuse in android applications,” *Proc. of the 2013 ACM SIGSAC Conf. on Computer & Comm. security - CCS ’13*, pp. 73–84, 2013.

[12] M. Georgiev, S. Iyengar, and S. Jana, “The most dangerous code in the world: validating SSL certificates in non-browser software,” in *Proc. of the 2012 ACM Conf. on Computer and Comm. security - CCS ’12 (2012)*, 2012, pp. 38–49.

[13] S. Fahl, M. Harbach, and T. Muders, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," *Proc. of the 2012 ACM Conf. on Computer and Comm. security*, pp. 50–61, 2012.

[14] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications," in *IEEE 12th Intn'l Conf. on Dependable, Autonomic and Secure Computing (DASC)*, 2014, pp. 75–80.

[15] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why Does Cryptographic Software Fail?: A Case Study and Open Problems," in *Proc. of 5th Asia-Pacific Workshop on Systems*, 2014, pp. 7:1–7:7.

[16] "Real World Cryptography Workshop Series." [Online]. Available: http://www.realworldcrypto.com.

[17] "The Heartbleed Bug." [Online]. Available: http://heartbleed.com/.

[18] "Apple's SSL/TLS 'Goto fail' bug." [Online]. Available: www.imperialviolet.org/2014/02/22/applebug.html.

[19] S. Vaudenay, "Security Flaws Induced by CBC Padding—Applications to SSL, IPSEC, WTLS...," *Advances in Cryptology—EUROCRYPT 2002*, no. 1, 2002.

[20] J. Rizzo and T. Duong, "Practical padding oracle attacks," *Proc. of the 4th USENIX Conf. on Offensive technologies (2010)*, pp. 1–9, 2010.

[21] T. Jager and J. Somorovsky, "How to break XML encryption," *Proc. of the 18th ACM Conf. on Computer and Comm. security - CCS '11*, p. 413, 2011.

[22] T. Duong and J. Rizzo, "Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET," *IEEE Symp. on Sec. and Priv.*, pp. 481–489, 2011.

[23] "Java Cryptography Architecture (JCA) Reference Guide." [Online]. Available: docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html.

[24] S. Fahl, M. Harbach, and H. Perl, "Rethinking SSL development in an appified world," *Proc. of the 2013 ACM SIGSAC Conf. on Computer & Comm. security - CCS '13 (2013)*, pp. 49–60, 2013.

[25] E. Bangerter, S. Krenn, M. Seifriz, and U. Ultes-Nitsche, "cPLC — A cryptographic programming language and compiler," *Information Security for South Africa*, pp. 1–8, Aug. 2011.

[26] M. Barbosa, A. Moss, and D. Page, "Constructive and Destructive Use of Compilers in Elliptic Curve Cryptography," *Journal of Cryptology*, vol. 22, no. 2, pp. 259–281, 2008.

[27] "Cryptol." [Online]. Available: http://www.cryptol.net.

[28] A. Bain, J. Mitchell, R. Sharma, and D. Stefan, "A Domain-Specific Language for Computing on Encrypted Data (Invited Talk).," *FSTTCS*, 2011.

[29] M. Barbosa, A. Moss, D. Page, N. F. Rodrigues, and P. F. Silva, "Type checking cryptography implementations," in *Fundamentals of Software Engineering*, Springer, 2012, pp. 316–334.

[30] M. Barbosa, D. Castro, and P. Silva, "Compiling CAO: From Cryptographic Specifications to C Implementations," *Principles of Security and Trust*, 2014.

[31] J. Lewis, “Cryptol: Specification, Implementation and Verification of High-grade Cryptographic Applications,” in *Proc. of the 2007 ACM Workshop on Formal Methods in Security Engineering*, 2007, p. 41.

[32] J. J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, “Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations,” in *ACM Conf. on Comp. & Comm. Sec. (SIGSAC)*, 2013, pp. 1217–1229.

[33] A. Moss, E. Oswald, D. Page, and M. Tunstall, “Automatic Insertion of DPA Countermeasures.,” *IACR Cryptology ePrint Archive*, vol. 2011, p. 412, 2011.

[34] A. Moss, E. Oswald, D. Page, and M. Tunstall, “Compiler assisted masking,” in *Cryptographic Hardware and Embedded Systems (CHES)*, 2012, pp. 58–75.

[35] G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi, “Compiler-based side channel vulnerability analysis and optimized countermeasures application,” in *Design Automation Conf. (DAC), 2013 50th ACM/EDAC/IEEE*, 2013, pp. 1–6.

[36] P. Bourque and R. Fairley, Eds., *Guide to the Software Engineering Body of Knowledge (SWEBOK)*, Version 3. IEEE Computer Society, 2014.

[37] A. Braga and E. Nascimento, “Portability evaluation of cryptographic libraries on android smartphones,” *Cyberspace Safety and Security*, pp. 459–469, 2012.

[38] D. González, O. Esparza, J. Muñoz, J. Alins, and J. Mata, “Evaluation of Cryptographic Capabilities for the Android Platform,” in *Future Network Systems and Security SE - 2*, vol. 523, Springer, 2015, pp. 16–30.

[39] P. Gutmann, “The design of a cryptographic security architecture,” *Proc. of the 8th USENIX Security Symposium*, 1999.

[40] D. Bernstein, T. Lange, and P. Schwabe, “The security impact of a new cryptographic library,” *Progress in Cryptology – LATINCRYPT 2012 (LNCS)*, vol. 7533, pp. 159–176, 2012.

[41] C. Forler, S. Lucks, and J. Wenzel, “Designing the API for a Cryptographic Library: A Misuse-resistant Application Programming Interface,” in *Proc. of the 17th Ada-Europe Intn’l Conf. on Reliable Software Technol.*, 2012, pp. 75–88.

[42] M. Fayad and D. C. Schmidt, “Object-oriented application frameworks,” *Comm. of the ACM*, vol. 40, no. 10, pp. 32–38, Oct. 1997.

[43] R. E. Johnson, “Frameworks = (components + patterns),” *Comm. of the ACM*, vol. 40, no. 10, pp. 39–42, Oct. 1997.

[44] A. D. A. Keromytis, J. L. J. Wright, T. De Raadt, and M. Burnside, “Cryptography As an Operating System Service: A Case Study,” *ACM Trans. Comput. Syst.*, vol. 24, no. 1, pp. 1–38, 2006.

[45] A. D. A. Keromytis, J. L. J. Wright, T. De Raadt, and T. De Raadt, “The Design of the {OpenBSD} Cryptographic Framework.,” in *USENIX Annual Technical Conf., General Track*, 2003, pp. 181–196.

[46] “Secure Coding Practices,” *OWASP*. [Online]. Available: https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide.

[47] SANS/CWE, “TOP 25 Most Dangerous Software Errors.” [Online]. Available: www.sans.org/top25-software-errors.

[48] “Cryptography Coding Standard.” [Online]. Available: cryptocoding.net/index.php/Cryptography_Coding_Standard.

[49] A. Lux and A. Starostin, “A tool for static detection of timing channels in Java,” *Journal of Cryptographic Engineering*, vol. 1, no. 4, pp. 303–313, Oct. 2011.

[50] A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, “Sleuth: Automated verification of software power analysis countermeasures,” in *Cryptographic Hardware and Embedded Systems-CHES 2013*, 2013, pp. 293–310.

[51] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, “CacheAudit: A Tool for the Static Analysis of Cache Side Channels,” *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, pp. 4:1–4:32, 2015.

[52] J. B. Almeida, M. Barbosa, J.-C. Filliâtre, J. S. Pinto, and B. Vieira, “CAOVerif: An open-source deductive verification platform for cryptographic software implementations,” *Science of Computer Prog.*, vol. 91, pp. 216–233, 2014.

[53] NIST, “Cryptographic Algorithm Validation Program (CAVP).” [Online]. Available: csrc.nist.gov/groups/STM/cavp/index.html.

[54] A. Braga and D. Schwab, “The Use of Acceptance Test-Driven Development to Improve Reliability in the Construction of Cryptographic Software,” in *The Ninth Intn’l Conf. on Emerging Security Information, Systems and Technologies (SECURWARE 2015). Accepted.*, 2015.

[55] OWASP, “Testing for Padding Oracle.” [Online]. Available: www.owasp.org/index.php/Testing_for_Padding_Oracle_(OTG-CRYPST-002).

[56] OWASP, “Top10 2013 (A6 - Sensitive Data Exposure).” .

[57] OWASP, “Key Management Cheat Sheet.” [Online]. Available: www.owasp.org/index.php/Key_Management_Cheat_Sheet.

[58] NIST, “Cryptographic Module Validation Program (CMVP).” [Online]. Available: http://csrc.nist.gov/groups/STM/cmvp/index.html.

[59] A. Barenghi, L. Breveglieri, I. Koren, D. Naccache, B. A. Barenghi, L. Breveglieri, I. Koren, F. Ieee, D. Naccache, and A. Barenghi, “Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures,” in *Proc. of the IEEE*, 2012, vol. 100, no. 11, pp. 3056–3076.

2.2.2 Practical Evaluation of Static Analysis Tools for Cryptography

This publication is entitled *"Practical Evaluation of Static Analysis Tools for Cryptography: Benchmarking Method and Case Study"* and was published at the 28th IEEE International Symposium on Software Reliability Engineering (ISSRE 2017), held in the city of Toulouse, France.

2017 IEEE 28th International Symposium on Software Reliability Engineering

Practical Evaluation of Static Analysis Tools for Cryptography: Benchmarking Method and Case Study

Alexandre Braga, Ricardo Dahab
Institute of Computing
State University of Campinas, Brazil
ambraga@cpqd.com.br, rdahab@ic.unicamp.br

Nuno Antunes, Nuno Laranjeiro, Marco Vieira
CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
nmsa@dei.uc.pt, cnl@dei.uc.pt, mvieira@dei.uc.pt

***Abstract*—The incorrect use of cryptography is a common source of critical software vulnerabilities. As developers lack knowledge in applied cryptography and support from experts is scarce, this situation is frequently addressed by adopting static code analysis tools to automatically detect cryptography misuse during coding and reviews, even if the effectiveness of such tools is far from being well understood. This paper proposes a method for benchmarking static code analysis tools for the detection of cryptography misuse, and evaluates the method in a case study, with the goal of selecting the most adequate tools for specific development contexts. Our method classifies cryptography misuse in nine categories recognized by developers (weak cryptography, poor key management, bad randomness, etc.) and provides the workload, metrics and procedure needed for a fair assessment and comparison of tools. We found that all evaluated tools together detected only 35% of cryptography misuses in our tests. Furthermore, none of the evaluated tools detected insecure elliptic curves, weak parameters in key agreement, and most insecure configurations for RSA and ECDSA. This suggests cryptography misuse is underestimated by tool builders. Despite that, we show that it is possible to benefit from an adequate tool selection during the development of cryptographic software.**

***Index Terms*—static analysis tools, cryptography, benchmarking, software security**

I. Introduction

Cryptography misuse is a common source of vulnerabilities introduced in software during development efforts [1], being mostly related to coding activities, but also found in design and architecture. Frequently, developers lack knowledge in applied cryptography, barely having support from experts when solving issues related to it.

Nowadays, many common software applications have strong security needs related to cryptography. For instance, mobile apps for secure end-to-end instant messages [2], authenticated encryption in SMS [3], and encrypted file systems with secure deletion [4], go far beyond traditional use cases for cryptography (e.g., SSL security, file encryption, or password protection) and require cryptographic protocols to be blended into their functionality. On the other hand, in ordinary software development, access to experts in applied cryptography is frequently restricted by tight schedules and limited budgets. Because these experts are rare, expensive, and busy, their involvement in software development is usually delayed to testing before deployment [5].

Static code analysis tools can help by detecting cryptography misuse during coding or reviews [6]. Such tools should be able not only to assist developers in daily coding activities, but also to support experts in efficiently finding potential issues during reviews and architecture analysis, according to structured methodologies [5]. A key point for the success of these methodologies is the correct choice of adequate static code analysis tools for specific development scenarios. For instance, in a simple scenario, ordinary developers, novice in cryptography, may need to code common use cases, requiring a precise, but not so complete, tool. In another scenario, knowledgeable developers may work together with cryptography experts to build sophisticated protocols blended into application functionality, requiring the most precise and complete toolkit available.

This paper proposes a benchmark for static code analysis tools (SCATs) focusing on their support for detecting cryptography misuse. Roughly speaking, our approach computes metrics (recall, precision, and f-measure) from measurements of tool execution on a set of test cases for cryptography misuse. The objective is twofold. First, to compare different tools directly, showing their limitations and strengths. Second, and more importantly, to determine how well tools perform in the context of cryptographic software development.

We have evaluated five free SCATs in order to answer the questions of how and to which extent cryptography misuse is caught by free SCATs currently available to developers. Free tools are readily available (no purchase of licences is required) and, in many cases, are the first and only option for ordinary developers. We found that the union of misuses detected by all five tools cover only about 35% of crypto misuses in our test cases. We also found that, in general, tools perform better in simple misuses regarding weak cryptography and bad randomness, and worse in issues for key management and program design flaws. Additionally, we found that the tools tested are unable to detect non-trivial misuses for insecure curve selection in Elliptic Curve Cryptography (ECC), weak parameters for key agreement with Diffie–Hellman (DH) and ECDH, misconfigured digital signatures with ECDSA, and many insecure configurations for RSA.

Our method uses a pondered sum of metrics (weighted metrics) to show how likely SCATs can detect expected cryptography misuse in certain application types. Also, such metrics capture development needs concerning SCATs, helping to assemble adequate toolkits for cryptographic software. We observed that weighted precision and weighted recall (in our method, the weighted versions of recall and precision) are better metrics for expressing tool strengths and limitations than non-weighted metrics. Finally, we recommend SCAT usages for specific scenarios of cryptographic software development, according to tool performance during benchmarking.

To the best of our knowledge, this work is the first practical evaluation of SCATs concerning support for cryptography usage,

2332-6549/17 $31.00 © 2017 IEEE
DOI 10.1109/ISSRE.2017.27

including a detailed set of test cases for cryptography misuse. The main contributions of this work are the following: (1) a method for SCAT evaluation and comparison in detecting cryptography misuse; (2) a set of realistic test cases for cryptography misuse in Java; (3) an assessment of free SCATs showing actual gaps in crypto misuse coverage; (4) the evaluation of the tools according to our method; and (5) recommendations for SCAT usage in specific development scenarios. Java was chosen for this instance of our method because this programming language has a well-known and stable cryptographic API and was adopted by the Android platform, being extensively used worldwide by ordinary developers.

The outline of this paper is as follows. Section II presents related work and Section III overviews our method. Section IV presents likely cryptography misuse for applications, Section V explains development contexts for cryptography, while Section VI describes benchmarking scenarios for cryptography. Section VII details the assessment results and Section VIII instantiates a benchmark as a case study. Section IX discusses our findings and analyses threats to validity. Section X concludes the paper.

II. Related work

This section presents related works on cryptography misuse, SCATs for cryptography, and metrics for evaluation of vulnerability detection tools.

A. Cryptography Misuse

Lazar et al. [7] show that only 17% of cryptography vulnerabilities are inside software libraries, the other 83% are misuse of libraries. Also, Chatzikonstantinou et al. [8] concluded that about 88% of Android apps misuse cryptography. Braga and Dahab [1] investigated the occurrence of cryptography misuse in on-line forums for cryptographic programming, showing that crypto misuses have high probabilities (e.g., 90% for Java).

For Egele et al. [9] and Shuai et al. [10], deterministic encryption is the most common misuse when a block cipher (e.g., AES or 3DES) uses the Electronic Code Book (ECB) mode. Asymmetric deterministic encryption with non-randomized RSA is also a misuse [11]. Hardcoded or repeated Initialization Vectors (IV) and hard-coded seeds for Pseudorandom Number Generators (PRNGs) are also frequent [9]. Other misuses come from exchanging operation modes without considering IV needs [1].

Georgiev et al. [12] and Fahl et al. [13] showed that libraries for SSL/TLS allow programmers to ignore parts of certificate validation, adding vulnerabilities exploitable by man-in-the-middle attacks. Recent studies showed misuses related to weak or misplaced parameters for RSA [14], key agreement misconfiguration (e.g., DH and ECDH) [15], and Elliptic Curve Cryptography (ECC) [16], [17] as well.

Nadi et al. [18] observed that developers usually implement simple use cases (e.g. user authentication, storage of login data, secure connections, and data encryption), but face difficulties when using low-level Java APIs. For instance, Shuai et al. [19] discovered that password protection in Android is greatly affected by cryptography misuse. Finally, Braga and Dahab [1] found that the most widespread misuse is weak cryptography, affecting several crypto use cases, and the most troublesome use case is encrypting data at rest, which is affected by several misuses.

B. Static Code Analysis for Cryptography

The way cryptography has been approached by Secure Software Development Lifecycles (SSDL) was recently studied [5], [6]. The conclusion is that, in general, development teams are aware of cryptography sensitivity, but: (i) they fail to adopt organized lifecycles intended to develop secure crypto software; and (ii) they lack expert help to assure quality at different stages of the development process. Furthermore, the development of secure crypto software still lacks professional tool support for most issues [6].

Domain-specific tools target the coding and testing of cryptographic algorithms and development of cryptographic libraries [6]. On the other hand, emerging research targets tools for development of cryptography-enabled functionality with established and standardized algorithms, as well as trusted libraries and frameworks, including tools for secure programming [20], [21] and verification [9], [10], [22], with particular interest in testing tools for crypto misuses [13], [23], [24].

Current coding standards [25], [26] do offer simple advice and general rules against cryptography misuse that could be automated by simple tools. Also, there are ordinary SCATs with simple rules for crypto misuses (e.g., [27]–[31]), providing late detection [32]. Highly specific issues (usually not detected by ordinary tools) have been addressed by academic prototypes, including SCATs for Android [9], [10], [13] and iOS [33] apps, Java code [21], and misuse of SSL libraries [22].

The Juliet test suite [34] offers synthetic examples of insecure coding and is part of a bigger set of test cases, provided by the Software Assurance Reference Dataset (SARD) [35]–[37], and used to evaluate SCATs [38]–[41]. However, these test suites are quite limited, being restricted to risky or broken encryption and hash functions (weak cryptography), and to deterministic PRNGs (bad randomness).

The open questions answered in this paper are how and to which extent free SCATs detect cryptography misuse.

C. Benchmarking Vulnerability Detection Tools

Benchmarking vulnerability detection tools has gained attention in areas where long-term security assurance is more important than detection of fancy vulnerabilities. Considering the evaluation of static code analysis tools in general, existing initiatives (such as the Software Assurance Metrics And Tool Evaluation - SAMATE [35], [42]) tried to establish a minimum set of test cases as baseline for vulnerability coverage. Recent initiatives [43] adopted a metrics-based approach in which general behavior for metrics (e.g., precision and recall) is more important than detection of specific vulnerabilities. Unfortunately, none of these initiatives focus on cryptography misuse.

Vulnerability detection tools classify parts of the target application in one of two classes: vulnerable or non-vulnerable. In this way, evaluation metrics are based on raw measures obtained from calculating a confusion matrix which represents the possible outcomes for each classified instance, according to a reference oracle of test cases. In Table I, True positives (TP) is the number of actual vulnerabilities detected by evaluated tools; False positives (FP) is the number of vulnerabilities detected that, in fact, do not exist (false alarms); False negatives (FN) is the number of true, but undetected vulnerabilities (omissions); and True negatives (TN) is the number of tests without actual vulnerabilities.

Antunes and Vieira [44] studied in depth the effectiveness of various metrics for benchmarking security tools and concluded that benchmarks for vulnerability detection tools should consider metrics that are able to capture the effectiveness of the tools in the concrete scenario where those tools are to be used. According to them [44], three metrics commonly used in benchmarks and easily computed from raw measures are Recall, Precision, and f-measure, but these may not be the best ones in all cases. Antunes and Vieira [45] also addressed the practical problem of benchmarking vulnerability detection tools in web services environments. They defined two benchmarks for vulnerability detection tools targeting the well-known SQL Injection vulnerability. One benchmark uses a predefined set of test cases, and the other is based on a user-defined selection of tests. Their results showed that the benchmarks accurately reflect the effectiveness of vulnerability detection tools. Their study, however, holds only for a single type of vulnerability.

TABLE I
CONFUSION MATRIX FOR BINARY CLASSIFICATION.

Oracle	**Reported by evaluated tool**	
Test case	**Condition Positive (P)**	**Condition Negative (N)**
Positive	True Positive (TP)	False Negative (FN)
Negative	False Positive (FP)	True Negative (TN)

Static code analysis of programs is undecidable in general [46]. In spite of that, many working solutions have been applied in practice, but in quite limited ways, as shown by several works that help understanding the limits of this technology. The following paragraphs analyze those related to ours.

Kupsch and Miller [47] compared manual and automated vulnerability assessment and found that from all the vulnerabilities discovered by manual assessment, tools found only simple implementation bugs, but did not find issues requiring deep understanding of code or design.

Diaz and Bermejo [38] compared tools against the SARD test suites [35] and analyzed results using known metrics (e.g., recall, precision, F-M), finding an average precision of 0.7 and an average recall of 0.527. They concluded [38] that it is not yet possible to standardized the behavior of SCATs, because the differences in design and technologies among current tools lead them to detect different kinds of vulnerabilities.

Mahonar et al. [48] assessed SCATs' ability to detect vulnerabilities in source code, concluding that it is hard to benchmark tools just by looking at the results produced against a certain test suite, confirming that tools find less flaws as the cyclomatic complexity of source code increases.

Goseva-Popstojanova and Perhinschi [39] used the Juliet test suite [34] to evaluate three commercial SCATs. They found [39] that 27% of C/C++ vulnerabilities and 11% of Java vulnerabilities were missed by all tools. They also found an overall recall, for each tool, close to or below 50%.

Delaitre et al. [40] compiled results from SAMATE's competitions using the SARD [35] data set. They concluded that, in general, design and coding clarity helps improving software assurance, because the simpler the control and data flow structure of the test case, the more effective the tools will be at finding weaknesses [40]. They also concluded that distinct SCATs generally do not find the same weaknesses [40], due to differences in tool design.

Shiraishi, Mohan, and Marimuthu [49] proposed evaluation criteria and new test suites for benchmarking SCATs. After evaluating commercial SCATs, they concluded that there are defects difficult to detect. This general purpose test suite does not tackle cryptography at all.

Finally, Hoole at al. [41] analyzed test suites from SARD [35], discussed various deficiencies and associated improvements, evaluating such improvements against SCATs. They showed that invalid assumptions in test design lead to imprecise measurement.

III. OVERVIEW OF THE BENCHMARKING METHODOLOGY

The methodology adopted in this work is based on that proposed by Antunes and Vieira [45]. We built our method around the concept of predefined, but realistic test cases [45]. The difference is that our test cases are specific for a whole domain (e.g., cryptography), instead of a single type of vulnerability in a broader domain. By coding test cases for types of bad practices found in an existing classification of cryptography misuse (described in Section III-B), we were able to capture the diversity of this domain. A **cryptography misuse** is a programming bad practice frequently found in cryptographic software [1], leading to vulnerabilities. We do not simply name them vulnerabilities because, in many cases, they are design flaws and insecure architectural choices.

The basic idea of our method is to exercise SCATs using sample programs for cryptography usage (with and without crypto misuses) and, based on the detected misuses, calculate a small set of metrics that capture the detection capabilities of SCATs in the cryptography domain. Also, we argue that, due to the many variables involved (e.g., crypto misuses, SCAT deployments, developer expertise in cryptography, the way misuses appear in applications, and availability of crypto experts), the definition of a benchmark for cryptography misuse detection tools must be specifically targeted to a particular scenario, in order to actually make choices about the benchmark components.

A. Benchmarking Procedure

Figure 1 illustrates the proposed benchmarking procedure. Its four stages are then described in detail.

1) **Planning**, where the benchmark is scoped and designed, using knowledge about cryptography misuse, with the following steps:
   a) determine likely misuses of cryptography for certain application profiles (refer to Section IV);
   b) determine the development context, including team experience and expert availability, and their relation with base metrics (Section V);
   c) determine the benchmarking scenario using combinations of weighted metrics for SCATs, configured according to profiles and contexts (Section VI).
2) **Preparation** consists in identifying and configuring tools according to a selected scenario for conducting a benchmarking campaign (Section VII).
3) **Operation** (Section VII), where measures are collected to further analysis and comparison, with the following steps:
   a) execution of the tools for the specified tests, detecting misuses;

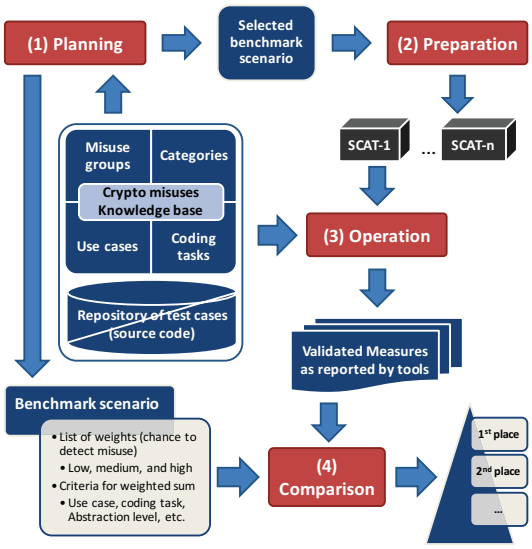


Fig. 1. Overview of the benchmarking procedure.

b) verification of raw measures (TP, FP, TN, and FN) by experts.

4) **Comparison** (Sections VII and VIII), where measurements are analyzed, interpreted, and prepared to presentation, with the following steps:
   a) normalization of results in order to compare the tools using common formats;
   b) computation of metrics from the raw measures;
   c) evaluation and ranking of the tools according to weighted metrics and scenarios.

The above procedure follows a general process for experimentation [50], customized for benchmarking of vulnerability detection tools [45]. The key novelty resides in the planning phase, which presents our contribution in systematically understanding how cryptography misuse affects development teams and tool selection.

B. Classification of Cryptography Misuses

Braga and Dahab [1] proposed a classification of cryptography misuses to capture how software developers actually misuse cryptography in practice. Their classification has nine categories: Weak Cryptography (WC), Bad Randomness (BR), Coding and Implementation Bugs (CIB), Program Design Flaws (PDF), Improper Certificate Validation (ICV), Public-Key Cryptography (PKC) issues, Poor Key Management (PKM), Cryptography Architecture and Infrastructure (CAI) issues, and IV/Nonce Management (IVM) issues. Each category is further detailed in more descriptive subsets in Table II.

The classification was obtained by an iterative and incremental process. First, the authors analyzed in different case studies [2]–[4] the pitfalls developers have to avoid to properly use cryptography. They also reviewed selected literature for software security that cover cryptography issues [51]–[56], and adopted their labels, grouping them in the main categories with few sub-items. Then, new categories were added and sub-items were refined from industry sources [57]–[59] and recent papers [7]–[10], [12], [13], [60]. Validation and further refinement occurred by studying online communities for cryptography programming [1].

In this work, we extend that classification to include other misuses and apply it to categorize our test cases. The main additions are insecure elliptic curves [17], repeated nonce in ECDSA [16], as well as misconfiguration of RSA [14] and DH [15]. The resulting classification (in Table II) assembles cryptography misuse in software collected from various sources: literature on software security (e.g., [51]–[56]), recent studies on cryptography misuse (e.g, [7]–[10], [12], [13], [60]), newly discovered misuses in Java (e.g., [14]–[17]), and industry initiatives for software security (e.g., [57]–[59]). Table II shows the grouping of misuse categories, misuse main categories, subsets, and sources.

IV. Application Profile and Misuse Groups

The application domain defines the cryptography requirements [5], which, in general, are satisfied by traditional use cases associated to cryptographic services [1], [18]: encrypting data at rest (EDR), secure communication (SC), password protection and encryption (PPE), authentication and validation of data (AVD), and digital rights management (DRM).

TABLE II
Crypto misuses from software security.

MG	Misuse category	Misuse subtype	Misuse source
Misuse Group 1 (MG1)	Weak Cryptography (WC)	- Risky or broken crypto - Proprietary cryptography - Determin. symm. encryption - Risky or broken hash/MAC - Custom implementation	[7] [8] [9] [10] [60] [51] [52] [53] [54] [55] [57] [58]
	Coding and Implementation Bugs (CIB)	- Wrong configs for PBE - Common coding errors - Buggy IV generation - Null cryptography - Leak/Print of keys	[7] [8] [9] [60] [19] [54] [58]
	Bad Randomness (BR)	- Use of statistic PRNGs - Predict., low entropy seeds - Static, fixed seeds - Reused seeds	[7] [8] [9] [60] [51] [52] [53] [54] [55] [57] [58]
Misuse Group 2 (MG2)	Program Design Flaws (PDF)	- Insecure default behavior - Insecure key handling - Insecure use of streamciphers - Insecure combo encrypt/auth - Insecure combo encrypt/hash - Side-channel attacks	[7] [8] [10] [11] [60] [51] [52] [55] [56] [57] [58]
	Improper Certificate Validation (ICV)	- Missing validation of certs - Broken SSL/TLS channel - Incomplete cert. validation - Improper validated host/user - Wildcards, self-signed certs	[7] [10] [12] [13] [60] [56] [58]
	Public-Key Cryptography (PKC) issues	- Deterministic encrypt. RSA - Insecure padding RSA enc. - Weak configs for RSA enc. - Insecure padding RSA sign. - Weak signatures for RSA - Weak signatures for ECDSA - Key agreement: DH/ECDH - ECC: insecure curves	[8] [9] [10] [11] [60] [14] [53] [55] [16] [17]
Misuse Group 3 (MG3)	IV and Nonce Management (IVM) issues	- CBC with non-random IV - CTR with static counter - Hard-coded or constant IV - Reused nonce in encryption	[8] [9] [10] [60]
	Poor Key Management (PKM)	- Short key, improper key size - Hard-coded or constant keys - Hard-coded PBE passwords - Reused keys in streamciphers - Use of expired keys - Key distribution issues	[7] [8] [9] [60] [51] [52] [53] [54] [55] [56] [59] [58]
	Crypto Architecture and Infrastructure (CAI) issues	- Crypto agility issues - API misunderstanding - Multiple access points - Randomness source issues - PKI and CA issues	[7] [8] [9] [10] [11] [13] [60] [51] [54] [55] [56] [57] [58]

These use cases are implemented in crypto coding tasks [1], [18]: encryption and decryption (Enc/Dec), digital signatures and verification (Sig/Ver), hashes or authentication codes and verification (Hash/Mac), key generation or agreement (KA), secure channels (e.g., SSL), digital certificate validation (Cert), and randomness generation (Rand). There are also secondary tasks, accessory to the main ones, representing operational (but important) aspects of crypto systems: key distribution, certificate generation, and key storage and recovery. In general, every crypto use case can be accomplished by a combination of coding tasks, where the complexity of the task is determined by the actual use case at hand.

Cryptography misuses are introduced by developers during coding tasks [1]. For this reason, an assessment setup for evaluation of cryptography usage during software development has to evaluate misuses according to their impact on coding tasks and related use cases. Cryptography misuses are not all equally difficult to avoid: some are easier to find and correct than others, depending on the level of abstraction (coding, design, architecture) required to identify the misuse. In our experience and related work [1], there are three qualitative groupings for the nine misuse categories (in Table II):

MG1 *(low complexity)* is related to coding activities and issues in APIs, and could be easily found by early detection techniques, simple code reviews, and skilled developers (supported by tools). It includes Weak Crypto (WC), Coding and Implementation Bugs (CIB), and Bad Randomness (BR). No deep understanding of program design is required, because fixes are likely to be simple and related to single programs or simple code snippets.

MG2 *(medium complexity)* is related to flaws in program design affecting a few different programs and may be difficult to identify due to feature distribution across programs. It includes Improper Certificate Validation (ICV) issues, Program Design Flaws (PDF), and Public-Key Crypto (PKC) issues. Fixes may require program redesign and may affect a few programs. Avoiding them requires more knowledgeable developers and support from experts.

MG3 *(high complexity)* is related to flaws in system design and architecture, and requires understanding of system architecture to analyze underlying cryptosystems. It includes Poor Key Management (PKM), IV and Nonce Management (IVM) issues, and Crypto Architecture and Infrastructure (CAI) issues. Fixes usually require new modules or redesign of modules, and may affect many code bases. These misuses require cryptography experts to perform code and design reviews, or architecture analysis.

Application's Crypto Misuse Profile determines likely crypto misuses in applications, as well as their complexity to be avoided by developers and likelihood to be detected by tools. A low-profile app benefits from early detection of misuses and is mainly associated to MG1; a high-profile app, associated with MG3, is better supported by late detection; and a medium-profile app, associated to MG2, is somewhere in between.

V. Development Contexts and Base Metrics

The *development context* determines the availability of cryptographic knowledge to development teams, either as actual developers with applied cryptography knowledge or as external consultants. There are four combinations of two binary variables: team **experience** (with or without experience) and expert **availability** (available or unavailable), resulting in the development contexts listed in Table III. There is a recommended application profile and misuse groups for each context.

TABLE III
Contexts combining teams and application profiles.

C#	Context	App profile	Misuse groups
C1	Novice team, no expert	Low complexity	MG1
C2	Novice team with expert	Low to medium	MG1 and MG2
C3	Skilled team, no expert	Medium to high	MG2 and MG3
C4	Skilled team and expert	High complexity	MG3

Benchmarking metrics are selected according to development contexts and app profiles in which SCATs will be used. Specific combinations of these factors define a benchmarking scenario. Table IV summarizes the metrics suitable for each development context (further discussed later for each context). We adopt the following definitions for recall, precision, and f-measure [44]. Recall is the proportion of positive cases that are correctly classified as positive and is computed by the formula $TP/(TP + FN)$. Precision is the proportion of the classified positive cases that are correctly classified and is computed by the formula $TP/(TP + FP)$. F-Measure represents the harmonic mean of precision and recall, given by the simplified formula $2 * TP/(2 * TP + FN + FP)$. Next subsections describe, for each development context, how teams interact with crypto experts, how crypto misuse shows up, how SCATs can be used, and which metrics are better to evaluate SCATs.

TABLE IV
Development contexts and recommended metrics.

Context	1st. Metric	2nd. Metric
C1	Precision	Recall
C2	Recall	Precision
C3	F-Measure	Recall
C4	Recall	Precision

A. Context 1: Unsupported Novice Team

This context comprises novice developers with little expert help, where novice developers want to avoid accessing (expensive or unavailable) crypto experts as much as possible, and solve simple issues by themselves. Accessing experts for false alarms produced by SCATs can be a waste of resources, but having expert help for double checking omissions is barely possible. When this novice team develops low-complexity apps, it may experience only simple misuses with code-based fixes, where no (or very few) program design is needed.

When this novice team develops medium-complexity apps, developers usually have to redesign a few programs when fixing misuses. In this case, SCATs are supposed to produce more false alarms. When the same novice team faces high-complexity apps (a barely recommended setup), developers usually suffer from complex misuses associated to system design and architecture. This context can result in software with the worst security (compared to other contexts) and is recommended only for low-complexity apps.

This way, a low false positive rate (FPR) is required for developers and a low false negative rate (FNR) helps optimize the time of experts. Also, precision is more important than recall, because experts will eventually be consulted later in verification and testing. This way, precision is the first metric for benchmarking SCATs and recall is a second option for tie breaking.

B. Context 2: Supported Novice Team

In this context, novice developers have access to crypto experts and want to use them as much as possible to overcome team's lack of skill in cryptography. Thus, accessing experts for SCATs false alarms or omissions is not a big deal. For low-complexity apps, the team has expert help for solving false alarms and omissions. When the novice team develops medium-complexity apps, expert review can be applied to remove false positives. When the same novice team faces high-complexity apps, developers can rely solely on cryptography experts to solve complex misuses, and SCATs are supposed to support the expert's work.

In this context, experts help developers to learn applied cryptography. This can result in better security and is good for medium- to high-complexity apps, depending on the actual availability of experts. It is also recommended for developments with loose schedules and low time pressure. As experts can always help with false positives and false negatives, despite the waste of time and delays, recall is the first metric for benchmarking, and precision is tie tiebreaker.

C. Context 3: Unsupported Knowledgeable Team

This comprises skilled developers acting as an autonomous team with rare expert help. In this context, experienced developers want to solve as many of the issues as possible, in a best effort approach, even if some issues are false positives. When faced with simple misuses, in low-complexity apps, this team is able to solve them quickly or recognize false alarms with a minimum waste of time.

On the other hand, when facing medium-complexity misuses, the time taken to discard false alarms may be relatively long, but still acceptable for a development cycle. In this setup, experts can be timely accessed to solve false alarms. In high-complexity apps, when complex misuses appear, developers may not be able to recognize all false alarms. Experts could quickly solve difficult issues, but they are not available for double checking omissions and false alarms. This context is suitable for medium-complexity apps. In this case, f-measure is the first metric and recall is the tiebreaker.

D. Context 4: Supported Knowledgeable Team

This context comprises a team with both skilled developers and crypto experts. Developers and experts work together to solve as many issues as possible, even if some of them are false alarms and omissions. In both low- and medium-complexity misuses, this team is able to solve issues quickly and recognize false alarms with minimum delay. When complex misuses appear, developers may not recognize all false alarms, but with expert help they can quickly solve difficult issues within affordable delays. The expert time taken to discard omissions may be relatively long, but still acceptable. This context can result in the best security and is suitable for high-complexity apps. It is also recommended for developments with tight schedules and high time pressure.

In summary, experts and developers deal together with false positives and false negatives. Also, a high FPR can be acceptable, because expert support and budget are available to solve them. In fact, it is important to maximize recall, but precision is not as important, because team members can easily identify false positives and plan for review in search of false negatives. In this context, recall is the first metric and precision is the tiebreaker.

VI. Benchmark Scenarios and Weighted Metrics

Benchmark scenarios define combinations of weighted metrics adopted to evaluate SCATs against likely misuses in development contexts. Like contexts, there are four main scenarios. The first scenario (S1) is a learner team with unavailable expert (C1) developing low-complexity apps, favoring simple misuses from MG1 (see Table III). The second (S2) is novice developers with highly available experts (C2), developing low- to medium-complexity apps, favoring MG1 and MG2. The third (S3) is an skilled team with unavailable expert (C3) that favors misuses from MG2 and MG3. The fourth scenario (S4) is skilled team with available experts (C4) that favors complex misuses from MG3. This section explains how the scenarios are built.

A. Metrics and Weights

The workload represents the work to be executed by SCATs during evaluation and is given by a number of test cases actually exercised. In our method, when evaluating tools for a specific scenario, all available test cases are included in the workload, and weights determine the importance of each misuse for the benchmark scenario. Direct metrics are computed for each misuse category (and for each tool under evaluation). Then, weighted metrics are computed by a weighted sum of direct metrics.

We adopt a qualitative approach to determine weight, which is derived by the expected performance of tools in each scenario. A misuse has greater weight if tools are expected to detect it in a specific scenario. Weights are not related to operational risk, chance of exploitation, or expected loss.

Table V shows the weights for four scenarios. For instance, in scenario one (S1), a SCAT should be adequate for novice developers, without expert help, developing low-complexity apps (C1). This SCAT should prioritize the detection of crypto misuses in MG1. This way, weights are defined as high for MG1 and low for both MG2 and MG3.

Weighted metrics for misuse categories can be computed by the formula $\sum_{i=1}^{9} w(i) * m(i) / \sum_{i=1}^{9} w(i)$, where $w(i)$ is the weight for misuse category i and $m(i)$ is the measured value for the metric in question (recall, precision, or f-measure). Weighted metrics for misuse groups, use cases, coding tasks, or any other criteria, can be computed accordingly. In a benchmark, it is necessary to choose a criteria. We adopted misuse categories for general tool evaluation, as well as misuse groups in our study case for tool benchmarking. Table VI shows four possible criteria to characterize misuses in a workload.

B. Workload Characterization

In our method, workload selection is not a big deal since all test cases are always included in the workload. Characterizing the workload is the important factor influencing metrics.

The workload was derived directly from the classification work by writing programs to exemplify misuses and their variants, for

TABLE V
WEIGHTS FOR SCENARIOS, CONTEXT, AND MISUSE GROUPS.

Scenario	Context	Weights per misuse group		
		MG1	MG2	MG3
S1	C1	High	Low	Low
S2	C2	Medium	High	Low
S3	C3	Low	Medium	High
S4	C4	Low	Low	High

TABLE VI
CRYPTO MISUSES DISTRIBUTED BY CATEGORIES.

Criteria	Subset	Misuse	Good use
Misuse group	MG1	61	34
	MG2	106	99
	MG3	35	49
Crypto use cases	EDR	90	93
	AVD	49	39
	RND	13	10
	PPE	10	5
	SC	40	35
Crypto coding tasks	Enc/Dec	83	68
	Sign/Ver	26	27
	Hash/MAC	22	11
	KG	43	61
	SSL	10	3
	Cert	5	2
	Rand	12	10
Misuse categories	WC	20	10
	CIB	29	16
	BR	12	8
	PDF	23	14
	ICV	15	5
	PKC/ENC	27	30
	PKC/SIG	21	25
	PKC/ECC	14	20
	PKC/KA	6	5
	IVM	8	10
	PKM	19	32
	CAI	8	7

each misuse category and its sub-items. Currently, our workload consists of 202 misuses (positives) and 182 good uses (negatives), with a total of 384 test programs for Java and its crypto API [61]. Good uses are the negative cases in the confusion matrix, provided by the testing oracle (see table I). We added expert knowledge to our workload by tagging test cases with the following criteria (used to sum up weights when computing weighted metrics):

- Crypto use cases and coding tasks (e.g., [1], [18]);
- Misuse groups for contexts and app profiles;
- Misuse categories (see Table II);
- Positives and negatives (see Table VI);
- Target platform (only Java [61] by the time of writing).

In order to avoid a large number of test cases in public-key crypto (PKC) issues, 68 positives and 80 negatives, we divided this category in four subsets for encryption with RSA, digital signatures with RSA and ECDSA, elliptic curve cryptography (for insecure curve selection), and key agreement with DH and ECDH, in Table VI. The number of test cases, according to different criteria, are shown in Table VI. All source code is public:

https://bitbucket.org/alexmbraga/cryptomisuses
https://bitbucket.org/alexmbraga/cryptogooduses

VII. ASSESSMENT OF SCATS FOR CRYPTOGRAPHY

This section describes the assessment of free SCATs with measures (TP, TN, FN, and FP) and metrics (precision, recall, and f-measure). We selected free SCATs from OWASP [62] and SAMATE [42]. By reading documentation, we identified five free SCATs for Java that also have rules for crypto issues: FindBugs 3.0.1 [63] with FindSecBugs 1.5.0 [27] (FSB), VisualCodeGrepper 2.1.0 [30] (VCG), Xanitizer 3.0.0 [29] (Xan), SonarQube 6.2 [28] with sonar-scanner 2.8 (SQ), and Yasca 3.0.5 [31]. All these tools perform late detection of vulnerabilities [32] and should be applied after developers have produced some source code.

Table VII puts together metrics for the five tools. They were computed uniformly for the workload (i.e., without considering the weights). Looking at precision and FP, a cursory glance would choose Yasca as the best tool, with higher precision and no FP. However, for the trained eye, Xanitizer would be a better choice, due to a higher recall and f-measure.

The tool with higher recall, Xanitizer (Xan), detected only one third of all misuses. The second higher recall (FSB) detected one quarter of misuses. Tools are not mutually exclusive and, in fact, have great intersection. When computing the union of TPs for all tools, we found a coverage of 35%, with 72 TPs (only 4 more than Xanitizer individually). These numbers are not good when compared to assessments with a broader scope [43], suggesting that free tools perform better in other security domains than in cryptography. Without a specific scenario to direct the comparison, f-measure seems to be the best metric to adopt, because it combines precision and recall. In this case, Xanitizer is the best choice.

TABLE VII
RESULTS FOR FIVE FREE SCATS.

Tools	Measures				Metrics		
	TP	TN	FN	FP	Prec.	Recall	F-M
FSB	51	147	151	35	0.593	0.252	0.354
Xan	68	140	134	42	0.618	0.337	0.436
SQ	5	181	197	1	0.833	0.025	0.048
VCG	7	180	195	2	0.778	0.035	0.066
Yasca	8	182	194	0	1.000	0.040	0.076

The following sections discuss the results in detail for misuse categories in the same abstraction level (MG1, MG2, and MG3).

A. Assessment of SCATs for Misuse Group 1

Table VIII shows metrics MG1. Concerning weak cryptography (WC), Xan and FSB seem to be the best tools with most TPs, but their FPs are high, resulting in low precision. SQ, VCG, and Yasca have no FP, resulting in high precision. However, these tools show low recall. Xan has the larger coverage, followed by FSB. Xan wins in recall and f-measure, being the best tool in this category.

Most tools detected DES and 3DES as insecure. Yasca did not detect 3DES. FSB and Xan detected weak hash functions (e.g., MD2, MD4, MD5, and SHA-1). Yasca did not detect SHA-1. Only Xan detected indirect references to weak hash functions. Also, only Xan detected an insecure MAC based on SHA-1. FSB and Xan detected insecure ECB mode. No tool detected Blowfish, RC4 or insecure PBE. No tool detected manual compositions of RSA and hashes. Most evaluated

SCATs behaved as simple pattern matchers for strings, thus, unable to detect misuses dependent of data-flow analysis.

For coding and implementation bugs (CIB), three tools did not score (no TP nor FP): SQ, VCG, and Yasca. Xan had most TP, but many false alarms (FP), obtaining the largest values for precision, recall, and f-measure. FSB got a second place, mostly influenced by a low recall. Both Xan and FSB detected buggy IV generation and NullCipher. Only Xan detected leaks of private or secret keys. However, leaks of DH's shared secrets were not detected. Tools did not detect saved keys in strings, misconfigured PBE, and concatenated inputs for hashes. VCG looked for the word "password" in comments, but misses other languages.

For bad randomness (BR), Table VIII shows that FSB, Xan, VCG, and Yasca have the highest precision with no FPs. However, VCG and Yasca got low recall. SQ did not score. Interestingly, FSB and Xan are tied in all three metrics (no tiebreaker). Also, Xan and FSB detected all uses of statistic PRNG and VCG missed a few misuses. Xan, FSB, and VCG detected only a few cases of low-entropy seeds. No tool detected fixed seeds nor reuse of seeds, suggesting the existence of blind spots in evaluated tools.

B. Assessment of SCATs for Misuse Group 2

Concerning program design flaws (PDF), Table IX shows that SQ, VCG, and Yasca did not score. FSB and Xan had the same value for recall, but FSB got the highest values for precision and f-measure. Taking precision as tiebreaker, FSB wins this category.

FSB and Xan detected insecure default for AES. FSB and Xan got insecure default for 3DES. Other insecure defaults (e.g., RSA, PBE, OAEP, and PRNG) were not detected. Tools have huge omissions in this category, showing many blind spots. No tool detected insecure combinations of encryption with hash or MAC. FSB and Xan detected some insecure stream ciphers. Side channels, such as timing channels in verification of hashes or MACs, and padding oracles, were not detected.

For public-key cryptography (PKC) issues, Table IX shows that VCG and Yasca did not score. FSB, Xan, and SQ got the highest precision with no FP. Xan wins this category due to higher recall and f-measure. Despite large TPs, the highest recall is still less than one quarter of all test cases for this category.

No tool detected insecure curves for ECC. Xan and FSB detected all cases of insecure padding for RSA and deterministic RSA. SQ got only one case. A bug in SQ makes it case sensitive for algorithm names, so that, for instance, two strings "RSA/NONE/NoPadding" and "RSA/None/NoPadding" are different and only the first one is detected. No tool detected insecure hashes or small parameters for RSA-OAEP. No tool detected small parameters for key agreement (e.g., DH or ECDH) and repeated message nonce for ECDSA. FSB and Xan detected only one case of weak configs for ECDSA (SHA1withECDSA), as well as only a few weak configs for RSA signatures. Tools are optimistic in this category to reduce FP, but having many omissions (FN). Also, tools are not updated with recent misuses, resulting in blind spots.

For improper certificate validation (ICV), according to Table IX, only FSB and Xan scored, with the same precision and no FP. FSB got better recall and f-measure for a higher TP, performing better in this category. Only FSB and Xan detected a few issues for certificate validation related to insecure SSL/TLS. FSB performed slightly better than Xan because of a bug in Xan that prevented the detection of misuses in nested classes. Again, tools limited themselves to just a few misuses in order to avoid mistakes.

C. Assessment of SCATs for Misuse Group 3

Concerning IV and nonce management (IVM) issues (see Table X), FSB has higher recall, but medium precision. FSB also got a higher value for f-measure, being the best choice for this category. VCG, SQ, and Yasca did not score. FSB and Xan detected constant IVs. No tool detected non-random IV for CBC, nor static counters for CTR. FSB detected one case of nonce reuse. The high number of FPs indicates tools have difficulty in understanding program design for IV management.

In poor key management (PKM), FSB and Xan got the same scores in all measures, resulting in a hard tie in all three metrics, as shown in Table X. VCG, SQ, and Yasca did not score in this category. FSB and Xan detected the same few cases of constant keys, constant passwords for PBE, and key reuse in stream ciphers. In this case, tool builders were less optimistic and tried to capture possible, but uncertain issues, resulting in more FPs due to lack of program understanding. Key management is made of design decisions hard to get from code analysis.

Considering Crypto Architecture and Infrastructure (CAI) issues, insecure architectural decisions are the most difficult cryptography misuse to detect by looking only at code. Only Xanitizer scored in this category, obtaining relatively good results for derived metrics. This happened because of a small number of test cases for one issue detected only by Xanitizer: the omission of a crypto provider during algorithm selection. However, this may be an FP, because crypto providers can be defined in configuration files as well.

VIII. Benchmarking Case Study

As an example, in this case study we select SCATs to be used by a novice team during development of an application with low-complexity crypto usage (context C1 in Table III). For illustrative purposes only, we suppose that the development process will target a hypothetical (but realistic) application for password protection and management, in which secure storage for passwords is encrypted by a master key derived from a master password. This simple app targets a quite common need in mobile devices [19]. This app has two main use cases: password protection with encryption (PPE) and Encrypting Data at Rest (EDR). Password integrity should also be guaranteed (AVD). A security assessment would show that its design and architecture are simple and key management is limited to a single master key/password. Development is likely to be affected by crypto misuses related to coding bugs (CIB), bad randomness usage (BR), password-based encryption misconfiguration (in CIB), and weak cryptography (WC).

The development team favors free tools and may only have part-time support of crypto experts, as external consultants. So we would like to select tools to promptly address MG1 issues, but also to point possible issues from MG2 and MG3, which will be eventually analyzed by an expert in the future. This way, the benchmark setup for this case study is that of scenario one (S1), consisting of development context one (C1 - Novice team, no expert), for a low-complexity app profile. In this setup, precision and recall are the adequate metrics, with recall as tiebreaker. Also, for computation of weighted metrics, misuse

TABLE VIII
RESULTS FOR MISUSE GROUP ONE (MG1).

Tools	Metrics for WC			Metrics for CIB			Metrics for BR		
	Prec.	Recall	F-M	Prec.	Recall	F-M	Prec.	Recall	F-M
FSB	0.727	0.40	0.516	0.50	0.172	0.256	1.0	0.417	0.588
Xan	0.588	0.50	0.541	0.70	0.483	0.571	1.0	0.417	0.588
SQ	1.0	0.20	0.333	0.0	0.0	0.0	0.0	0.0	0.0
VCG	1.0	0.20	0.333	0.0	0.0	0.0	1.0	0.250	0.400
Yasca	1.0	0.30	0.462	0.0	0.0	0.0	1.0	0.167	0.286

TABLE IX
RESULTS FOR MISUSE GROUP TWO (MG2).

Tools	Metrics for PDF			Metrics for PKC			Metrics for ICV		
	Prec.	Recall	F-M	Prec.	Recall	F-M	Prec.	Recall	F-M
FSB	0.417	0.217	0.286	1.0	0.221	0.361	1.0	0.267	0.421
Xan	0.357	0.217	0.270	1.0	0.235	0.381	1.0	0.133	0.235
SQ	0.0	0.0	0.0	1.0	0.015	0.029	0.0	0.0	0.0

TABLE X
RESULTS FOR MISUSE GROUP THREE (MG3).

Tools	Metrics for IVM			Metrics for PKM			Metrics for CAI		
	Prec.	Recall	F-M	Prec.	Recall	F-M	Prec.	Recall	F-M
FSB	0.286	0.500	0.364	0.263	0.263	0.263	0.0	0.0	0.0
Xan	0.231	0.375	0.286	0.263	0.263	0.263	0.800	1.0	0.889

TABLE XI
WEIGHTED METRICS FOR FIVE SCATS IN FOUR SCENARIOS.

Tools	Weighted metrics for S1			Weighted metrics for S2			Weighted metrics for S3			Weighted metrics for S4		
	W-Prec.	W-Recall	W-F-M	W-Prec.	W-Recall	W-F-M	W-Prec.	W-Recall	W-F-M	W-Prec.	W-Recall	W-F-M
Xan	**0.737**	0.451	0.537	0.756	**0.302**	0.392	0.563	0.431	**0.427**	0.488	**0.510**	0.471
FSB	0.701	0.316	0.425	0.747	0.266	0.377	0.412	0.253	0.270	0.281	0.259	0.242
Yasca	0.556	**0.130**	0.208	0.208	**0.049**	0.078	0.042	0.010	**0.016**	0.056	0.013	0.021
VCG	0.556	0.125	0.204	0.208	0.047	0.076	0.042	0.009	0.015	0.056	0.013	0.020
SQ	0.306	0.056	0.093	0.313	0.024	0.041	0.125	0.006	0.010	0.056	0.006	0.010

groups MG1, MG2, and MG3 are weighted as high, low, and low, respectively (see Table V).

Table XI puts together weighted metrics for the five evaluated tools from Section VII in the four scenarios (sorted by scenario one, the one of interest in the current example). Weighted metrics were computed as weighted sums of metrics calculated for misuse categories and weights associated to misuse groups, as discussed before.

For scenario one, taking only weighted precision as reference, Xan is the best choice. By giving a higher weight to MG1, we actually amplified the importance of simple misuses, where Xanitizer scored slightly better than FSB and much better than other tools. Table XI shows three groups of scores for scenario one (S1): Xan and FSB with high scores, VCG and Yasca with medium scores, and SQ with a low score. These numbers indicate that Xan and FSB not only performed better in general, but also preferred simple misuses from MG1, confirming what we saw in Section VII: Xan and FSB (individually) have more rules for crypto than all other tools together and most of them are for MG1.

The unbalanced number of rules among SCATs favored Xan and FSB. Table XI shows that Xan and FSB easily ranked in first and second places in all scenarios. On the other hand, the competition for ranking in third place was not that ease. For scenario one (S1), Yasca got third place thanks to tie breaking in recall. In scenario two (S2), where recall is the first metric, Yasca performed better and got third place, despite being tied with VGC and performed worst than SQ in precision.

For scenario three (S3), where f-measure rules and recall is the tiebreaker, Yasca got third place with a tight difference from other two SCATs. For scenario four (S4), we found not only the smallest metrics in general, but also the smaller differences among weighted metrics for Yasca, VCG, and SQ. Yasca and VCG were tied in both preferred metrics for S4, so the contend was solved by f-measure in favor of Yasca with tine advantage.

Evaluated SCATs favor scenario one because of a technology bias found in free SCATs toward low-complexity misuses, as discussed in Section VII. When we forced the use of free SCATs in other scenarios, their performance reduces gradually as misuse complexity increases. This behavior is perceived in Table XI by the progressively reduced values of weighted metrics (from scenarios one to four).

A. Misuse Groups and Tool Behavior

An optimistic tool only alerts about clear misuses and keeps silent about dubious ones. On the other hand, a pessimistic tool warns about every suspected misuse, even unlikely ones. SCATs

for security have to be pessimistic in order to avoid dangerous omissions [53]. On the other hand, SCATs should favor early detection of vulnerabilities in order to benefit from developer's short-term memories when fixing vulnerable code [32].

Based upon our observations and findings, we generalized the following expected behaviors for crypto-friendly, pessimistic SCATs (Table XII). Tools can be quite precise in early detection of MG1 (WC, CIB, and BR), showing relatively few FPs. Recall inside the misuse group (group recall) is expected to be relatively high, with few FNs. However, non-detected misuses from MG2 and MG3 cause dangerous omissions (FNs in overall recall).

In MG2 (ICV, PDF, and PKC), tools are expected to be less precise, producing more false alarms and omissions than in MG1. This is expected due to FPs caused by incomplete understanding of program design, as well as FNs due to misconceptions about programs. In MG2, optimistic tools are expected to show relatively low recall and many FNs, while pessimistic tools are expected to have relatively low precision and many FPs. These tools are better suited to late detection of crypto misuses.

In MG3 (PKM, IVM, and CAI), tools are expected to be quite imprecise, producing more false alarms and omissions than in other misuse groups. This behaviour is expected because of FPs due to partial understanding of software architectures, as well as FNs due to misconceptions about program design. Optimistic tools will have relatively low recall, while pessimistic tools will have lower precision. Tools for MG3 are better suited to late detection of misuses.

As discussed previously, all free SCATs in this benchmark favor MG1 and gradually decrease their performance in (weighted) metrics for MG2 and MG3. Table XII summarizes the relation between misuse groups and tool behavior.

TABLE XII
EXPECTED TOOL BEHAVIOR FOR CRYPTO-FRIENDLY SCATS.

Misuse group	Tool behavior
MG1	Higher precision and high recall
MG2	Low precision and moderate recall
MG3	Lower precision and high recall

B. Use of SCATs for Cryptography

Knowing the limitations of tools is important to understand how to use them more effectively. Table XIII summarizes tool usages for misuse groups. First, the recommended usage of SCATs for MG1. By precisely detecting many code-based misuses, SCATs can be integrated to IDEs and provide real-time support to developers during coding tasks (e.g., early detection). Since precision is high, these tools will need less expert supervision in coding. Because overall recall is low, tools alone will result in low quality (less secure) software, which can be subjected to further verification. This tool usage is recommended for development context 1 (novice teams and barely available experts).

The recommended tool usage for MG2 follows. SCATs should not be the only way to find design flaws in coding, because design flaws are supposed to be found earlier in development, before coding tasks. If they are being found in coding, this may be a symptom of expert unavailability or team amateurism. Tools can be applied during system integration (daily or weekly build) and support design reviews or manual inspection, possibly later in system development. This tool usage is recommended for novice teams with expert help (context 2) or skilled teams and barely available experts (context 3).

As for MG3, SCATs should not be the only way to find architectural flaws in coding, because they are supposed to be found very early in development. If they are not, this may be a symptom of expert unavailability or team building issues (e.g., lack of security architects). Extensive use of tools during coding tasks is inadequate and can mistakenly divert team effort to correct nonexistent misuses or correct existing, but complex, misuses the wrong way. This tool usage is better for supporting experts during manual inspections in architectural security analysis and is recommended for development contexts where crypto experts are always available (e.g., contexts two and four).

Considering the evaluated tools, FindSecBugs can be integrated to IDEs and building tools. VisualCodeGrepper (VCG) has both CLI and GUI, and its CLI can be called by scripts in building processes, but not integrated to IDEs. Xanitizer can be used as a standalone tool with its own GUI and also be incorporated in a build process. SonarQube can be integrated to building tools and claims to have plugins for many IDEs.

Finally, the current version of the best ranked tool in this benchmark (Xanitizer) is not easily integrated to IDEs. The second option (FindSecBugs) is better suited for late detection of misuses within IDEs. Also, Xanitizer and FindSecBugs are adequate for contexts one and two, with the drawback of showing relatively more false positives and false negatives in the second case. We do not believe any of evaluated tools is adequate for contexts three and four, due to their poor performance in this benchmark.

TABLE XIII
CONTEXTS LINK LIKELY MISUSES AND TOOL USAGE.

Context	Misuse group	Usage	Tool
C1	MG1	Integrated to IDE	Xan and FSB
C2	MG1 and MG2	IDE and build	Xan and FSB
C3	MG2 and MG3	Build and review	None
C4	MG3	Reviews	None

IX. DISCUSSION AND THREATS TO VALIDITY

This section discusses limitations that threat validity, and other general aspects of our benchmarking methodology.

A. Limitations and Threats to Validity

We are aware that our workload is a sample of all possible test cases for cryptography misuse. For instance, we did not implement all variations of algorithm names in case-sensitive strings. Also, we explicitly avoided test cases for specific technologies and platforms (e.g., web, mobile, etc.).

We found a general coverage of around 35%, with the two best scores around 33% and 25%. These numbers are not surprising. For instance, Antunes and Vieira [45] concluded in their benchmark that the effectiveness of tools is poor. Also, OWASP's benchmark [43] of tools for web security shows a coverage of 50% for the best tool. Other works [38], [39] have similar results. These arguments suggest that tools perform better in other security domains than in cryptography.

Most evaluated tools use pattern matching as the main technique for vulnerability detection. In particular,

VisualCodeGrepper, SonarQube, and Yasca seem to use only simple pattern matching. FindSecBugs detects some non-trivial patterns, such as zeroed IVs and insecure interface implementations. Xanitizer uses taint analysis to detect simple leaks of keys and indirect references to weak algorithms.

Our workload is made of realistic test cases, meaning that they are not artificial, but they are not real applications either. They were collected and derived from crypto misuse patterns found in many sources. Still, they may contain crypto misuses not frequently found in real applications. On the other hand, the diversity provided by our test cases is hard to obtain by only using actual applications.

When loading test cases into tools, auxiliary code had to be excluded from the workload. Otherwise, evaluated tools would consider them in vulnerability scans, generating even more false positives. A simple and fast solution is to exclude auxiliary code from raw measures.

Our test cases target crypto misuse and neglect other aspects of code quality (including secure coding) in order to keep programs small, self-contained, and self-explained. However, these design decisions were not captured by tools. Thus, when validating raw measures, we had to exclude issues not related to crypto misuses.

Crypto misuses can appear in combinations of two or three, showing a strong relation to each other [1]. Because of that, some test cases had to include more than one misuse, in order to capture the intended misbehavior. In these cases, we chose to ignore auxiliary misuses (when detected by tools) and focus on detection of main misuses. Auxiliary misuses have their own test cases.

When evaluated tools had no default configuration that included verification of crypto misuses, we always preferred the more complete (full) scan. Also, for output, we preferred reports with all severity classes (levels) enabled. This practice was adopted to avoid complicated settings and customized configurations, which can be hard to reproduce.

B. General Discussion

Cryptography misuse is not related to the implementation of cryptographic algorithms. Instead, crypto misuses emerge when ordinary developers use cryptographic libraries (APIs) in their daily coding activities. These developers can also use SCATs to receive detailed reports about issues and how to correct them.

This work sits in the intersection of three domains (e.g., applied cryptography, software engineering, and software security) sharing a common knowledge base. We believe that our experiment actually tests what we meant to test, because tool builders have codified this knowledge (to some degree) in SCATs, which are able to detect cryptography misuse, even in limited ways.

SCATs generate detailed reports that have to be manually inspected. By inspecting these reports, we were able to identify all issues related to cryptography, separating them from other issues, and classifying them as true positives and false positives. We mitigated the threat of mislabeling misuses, because crypto misuses were clearly identifiable due to the systematic use of our classification to generate test cases and our experience in this kind of code review.

Finally, our conclusions are based upon a repeatable experiment and are consistent with related work and our previous results. In fact, our numbers were consistent among executions, because evaluated SCATs are deterministic and our benchmarking method is reproducible. Previous results were obtained from case studies, an extensive literature review, and empirical studies of online communities.

X. Conclusion

Cryptography usage is full of design decisions that defy early detection of misuse. Also, coverage of cryptography misuse by tools is far from good, with current tools showing many blind spots. We saw a huge gap between what experts actually see as cryptography misuse and what tools currently detect. Therefore, expert help is required to assure quality in different moments of development efforts. We argue that, with current tools' maturity, an adequate toolkit has to be carefully crafted to fit the needs of teams for specific development contexts.

Our methodology was able to distinguish among tools for distinct usage scenarios and find the best suited options, despite limitations in current SCAT technology. Also, the method provided the necessary context to analyze the reasons why evaluated tools are not suitable for advanced scenarios.

As future work, in the short term, we plan to improve test cases to include new misuses, variations of existing ones, as well as perform assessments of commercial-off-the-shelf tools. In the long run, we plan to expand test cases to other programming languages, software platforms, cryptographic libraries, as well as dynamic analysis tools. Finally, we foresee the emergence of a new generation of tools able to better assist the development of secure cryptographic software.

Acknowledgment

This work has been partially supported by CNPq, Intel, and the projects **DEVASSES** (www.devasses.eu), funded by the European Union's FP7 under grant agreement no. PIRSES-GA-2013-612569, and **EUBra-BIGSEA** (www.eubra-bigsea.eu), funded by the European Commission under Horizon 2020 grant agreement no. 690116. R. Dahab and A. Braga acknowledge the support of Fapesp's thematic project #2013/25.977-7. Authors also thank UNICAMP, CPqD, and University of Coimbra for the institutional support.

References

[1] A. Braga and R. Dahab, "Mining Cryptography Misuse in Online Forums," in *2nd International Workshop on Human and Social Aspect of Software Quality*, 2016.

[2] A. Braga and D. Schwab, "Design Issues in the Construction of a Cryptographically Secure Instant Message Service for Android Smartphones," in *The 8th Inter. Conf. on Emerging Security Information, Systems and Technologies*, nov 2014, pp. 7–13.

[3] A. Braga, R. Zanco Neto, A. Vannucci, and R. Hiramatsu, "Implementation Issues in the Construction of an Application Framework for Secure SMS Messages on Android Smartphones," in *The 9th Inter. Conf. on Emerging Security Information, Systems and Technologies*. IARIA, 2015, pp. 67–73.

[4] A. Braga and A. Colito, "Adding Secure Deletion to an Encrypted File System on Android Smartphones," in *The 8th Inter. Conf. on Emerging Security Information, Systems and Technologies*, nov 2014, pp. 106–110.

[5] A. Braga and R. Dahab, "Towards a Methodology for the Development of Secure Cryptographic Software," in *The 2nd International Conference on Software Security and Assurance (ICSSA 2016)*, 2016.

[6] ——, "A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software," in *XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg 2015)*, Florianópolis, SC, Brazil, 2015, pp. 30–43.

[7] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why Does Cryptographic Software Fail?: A Case Study and Open Problems," in *5th Asia-Pacific Workshop on Systems*, ser. APSys '14. New York, NY, USA: ACM, 2014, pp. 7:1–7:7.

[8] A. Chatzikonstantinou, C. Ntantogian, C. Xenakis, and G. Karopoulos, "Evaluation of Cryptography Usage in Android Applications," *9th EAI International Conference on Bio-inspired Information and Communications Technologies*, 2015.

[9] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," *ACM SIGSAC conference on Computer & comm. security (CCS'13)*, pp. 73–84, 2013.

[10] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications," in *IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC)*, 2014, pp. 75–80.

[11] P. Gutmann, "Lessons Learned in Implementing and Deploying Crypto Software," *Usenix Security Symposium*, 2002.

[12] M. Georgiev, S. Iyengar, and S. Jana, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*, 2012, pp. 38–49.

[13] S. Fahl, M. Harbach, and T. Muders, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," in *ACM conference on Computer and communications security*, 2012, pp. 50–61.

[14] E. S. Alashwali, "Cryptographic vulnerabilities in real-life web servers," in *Third International Conference on Communications and Information Technology (ICCIT)*. IEEE, jun 2013, pp. 6–11.

[15] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, and Others, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 5–17.

[16] J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow, "Elliptic curve cryptography in practice," in *Financial Cryptography and Data Security*. Springer, 2014, pp. 157–175.

[17] V. G. Mart and L. Hern, "Implementing ECC with Java Standard Edition 7," *International Journal of Computer Science and Artificial Intelligence*, vol. 3, no. 4, pp. 134–142, 2013.

[18] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, ""Jumping Through Hoops": Why do Java Developers Struggle With Cryptography APIs?" *The 38th International Conference on Software Engineering*, 2016.

[19] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Analysis on Password Protection in Android Applications," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on*, nov 2014, pp. 504–507.

[20] S. Fahl, M. Harbach, and H. Perl, "Rethinking SSL development in an appified world," *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, pp. 49–60, 2013.

[21] S. Arzt, S. Nadi, K. Ali, E. Bodden, S. Erdweg, and M. Mezini, "Towards Secure Integration of Cryptographic Software," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2015)*. New York, NY, USA: ACM, 2015, pp. 1–13.

[22] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting SSL usage in applications with SSLint," in *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015, pp. 519–534.

[23] OWASP, "OWASP Testing Project," 2015. [Online]. Available: https://www.owasp.org/index.php/OWASP_Testing_Project

[24] J. Rizzo and T. Duong, "Practical padding oracle attacks," *Proc. of the 4th USENIX conf. on offensive technologies (2010)*, pp. 1–9, 2010.

[25] OWASP, "OWASP Top Ten Project," 2013. [Online]. Available: https://www.owasp.org/index.php/Top_10

[26] SANS/CWE, "TOP 25 Most Dangerous Software Errors." [Online]. Available: www.sans.org/top25-software-errors

[27] P. Arteau, "FindSecBugs." [Online]. Available: https://find-sec-bugs.github.io

[28] SonarSource, "SonarQube." [Online]. Available: https://www.sonarqube.org

[29] RigsIT, "Xanitizer." [Online]. Available: https://www.rigs-it.net

[30] NCCGroup, "VisualCodeGrepper." [Online]. Available: https://github.com/nccgroup/VCG

[31] M. Scovetta, "Yasca." [Online]. Available: http://yasca.org

[32] L. Sampaio and A. Garcia, "Exploring context-sensitive data flow analysis for early vulnerability detection," *Journal of Systems and Software*, vol. 113, pp. 337 – 361, 2016.

[33] Y. Li, Y. Zhang, J. Li, and D. Gu, "iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications," in *8th International Conference on Network and System Security*. Xi'an, China: Springer International Publishing, 2014, pp. 349–362.

[34] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012.

[35] NIST, "Software Assurance Reference Dataset (SARD)." [Online]. Available: https://samate.nist.gov/SRD/

[36] P. E. Black, "Counting bugs is harder than you think," in *11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2011, pp. 1–9.

[37] P. Black, "Static analyzers: Seat belts for your code," *IEEE Security & Privacy*, vol. 10, no. 3, pp. 48–52, 2012.

[38] G. Díaz and J. R. Bermejo, "Static analysis of source code security: Assessment of tools against SAMATE tests," *Information and Software Technology*, vol. 55, no. 8, pp. 1462–1476, 2013.

[39] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology*, vol. 68, pp. 18–33, 2015.

[40] A. Delaitre, B. Stivalet, E. Fong, and V. Okun, "Evaluating bug finders - Test and measurement of static code analyzers," *Proceedings - 1st International Workshop on Complex Faults and Failures in Large Software Systems, COUFLESS 2015*, pp. 14–20, 2015.

[41] A. M. Hoole, I. Traore, A. Delaitre, and C. de Oliveira, "Improving Vulnerability Detection Measurement: [Test Suites and Software Security Assurance]," *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pp. 27:1–27:10, 2016.

[42] NIST, "Software Assurance Metrics And Tool Evaluation." [Online]. Available: https://samate.nist.gov

[43] OWASP, "OWASP Benchmark Project." [Online]. Available: https://www.owasp.org/index.php/OWASP_Benchmark_Project

[44] N. Antunes and M. Vieira, "On the Metrics for Benchmarking Vulnerability Detection Tools," in *The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*. Rio de Janeiro, Brazil: IEEE, 2015.

[45] ——, "Assessing and Comparing Vulnerability Detection Tools for Web Services: Benchmarking Approach and Examples," *IEEE Transactions on Services Computing*, vol. 8, no. 2, pp. 269–283, 2015.

[46] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.

[47] J. A. Kupsch and B. P. Miller, "Manual vs. automated vulnerability assessment: A case study," in *First International Workshop on Managing Insider Security Threats (MIST)*, 2009, pp. 83–97.

[48] L. Manohar, R. Velicheti, D. C. Feiock, M. Peiris, R. Raje, and J. H. Hill, "Towards Modeling the Behavior of Static Code Analysis Tools," *2014 9th Cyber and Information Security Research Conference*, 2014.

[49] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 12–15, 2015.

[50] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[51] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, 2001.

[52] M. Howard and D. LeBlanc, *Writing Secure Code*, 2nd ed. Redmond, Wash: Microsoft Press, Dec. 2004.

[53] B. Chess and J. West, *Secure programming with static analysis*, 2007.

[54] M. Howard, D. LeBlanc, and J. Viega, *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill Education, 2009.

[55] M. Howard and S. Lipner, *The Security Development Lifecycle*. Redmond, WA, USA: Microsoft Press, 2006.

[56] A. Shostack, *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[57] Safecode, "Fundamental Practices for Secure Software Development," 2011. [Online]. Available: http://www.safecode.org/wp-content/uploads/2014/09/SAFECode_Dev_Practices0211.pdf

[58] OWASP, "Cryptographic Storage Cheat Sheet." [Online]. Available: www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet

[59] CYBSI, "Avoiding The Top 10 Software Security Design Flaws," 2014. [Online]. Available: http://cybersecurity.IEEE.org/

[60] A. Braga and R. Dahab, "Introdução à Criptografia para Programadores: Evitando Maus Usos da Criptografia em Sistemas de Software," in *Caderno de minicursos do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2015*, 2015, pp. 1–50.

[61] Oracle, "Java Cryptography Architecture (JCA) Reference Guide." [Online]. Available: docs.oracle.com/javase/8/docs/technotes/guides/\security/crypto/CryptoSpec.html

[62] OWASP, "List of Source Code Analysis Tools." [Online]. Available: https://www.owasp.org/index.php/Source_Code_Analysis_Tools

[63] UMD, "FindBugs." [Online]. Available: http://findbugs.sourceforge.net

2.3 Development methods for secure crypto software

This section contains the following publications. First, the publication entitled *"Towards a Methodology for the Development of Secure Cryptographic Software"*. Second, the publication entitled *"Understanding the Field of Cryptographic Software Security"*.

2.3.1 Towards a Methodology for the Development of Secure Cryptographic Software

This publication is entitled *"Towards a Methodology for the Development of Secure Cryptographic Software"* and was published at the 2nd. International Conference on Software Security and Assurance (ICSSA 2016). This paper was awarded the best student paper at that conference, held in the city of St. Polten, Austria.

2016 International Conference on Software Security and Assurance

Towards a Methodology for the Development of Secure Cryptographic Software

Alexandre Braga and Ricardo Dahab
Institute of Computing
State University of Campinas (UNICAMP)
Campinas, Brazil
e-mail: ambraga@cpqd.com.br, rdahab@ic.unicamp.br

***Abstract*—Historically, software security has approached the development of cryptographic software merely as a feature to be added. This stance did not have a positive influence on the design of advanced security functionalities into modern software. Thus, this work proposes a methodology for development of secure cryptographic software, providing a structured way to approach cryptography into secure development methods. Our methodology captures the (otherwise casual) practices of secure software development employed by modern software factories when building security-critical software with cryptographic technologies. In time, we compare our work with other methods, provide actual examples, as well as analyze architectural aspects and specialized tool support for secure coding and verification of cryptographic software. We do believe this work contributes to bridge the gap between modern cryptographic software and today's development methods for secure software.**

Keywords-cryptography; software security; secure software development; highly adaptive lifecycles; software assurance

I. INTRODUCTION

Today's most popular cryptographic software are massively available, cloud-based mobile apps that perform secure, peer-to-peer communication, with security requirements strongly related to their advanced services. However, in general, we observed a lack of methods for building secure cryptographic software in modern software factories, in such a way that cryptographic security could be easily blended to functionality. Also, historically, practitioners reserved little attention to cryptography misuses, compared to other vulnerabilities.

The main contribution of this work is to structure a methodology for Development of Secure Cryptographic Software (DSCS), which provides an ordered way to approach cryptography into Secure Software Development Lifecycles (SSDL). DSCS adopts software architecture aspects, foster specialized tool support, and can enhance the ability of other SSDL methods to handle cryptography issues, even with highly adaptive lifecycles. Furthermore, DSCS is supported by practical evidence of cryptography misuse as well as generalizes several practices we have seen and applied in actual software developments.

The text is organized as follows. Section II gives background and motivation. Section III analyzes current practice for cryptography in SSDL. Section IV deeply describes our methodology. Section V concludes the text.

II. BACKGROUND AND MOTIVATION

In this text, cryptographic software (crypto software, for short) is software that preserves major security goals (namely, confidentiality, integrity, authenticity, and non-repudiation) transparently blended into functionality, by using cryptographic methods available through reusable libraries and frameworks. Applied cryptography is the use of cryptographic infrastructures (e.g., packages, libraries, and APIs) to build cryptographic security into security-critical software.

For instance, the following security-critical apps go far beyond traditional use cases for cryptography (e.g., SSL security, file encryption, or password protection) and require cryptographic protocols to be blended to (or co-designed with) app's functionality, exemplifying our vision of modern crypto software, as well as DSCS principles and ideas: secure end-to-end instant messages [1], authenticated encryption for SMS [2], encrypted file systems with secure deletion [3], and frameworks for cryptographic security on mobile devices [4].

The confidence in secure software is always relative to the assurance methods in use [5]. Nevertheless, in general, secure software engineering [5][6] seems not to directly address the issues of cryptographic security. In fact, studies have shown that vulnerabilities in crypto software have been mainly caused by software defects and poorly managed parameters [7]–[10]. Recent studies [11]–[15] showed the occurrence of known cryptography misuses. Sadly, current methods are unable to cope with security issues in programming crypto features [16][17], leading to cryptography frequent misuse [11][14][15], improper certificate validation [12][13]; and inappropriate error handling when orchestrating crypto services [18][19].

III. DECONSTRUCTING TRADITIONAL PRACTICE

This section illustrates our viewpoint by reviewing cryptography's occurrence into software security literature. Naturally, this review is illustrative rather than exhaustive.

Cryptography texts include books [20][21] intended for scientists and engineers embedding cryptography into hardware or implementing it in software; also, there are books on cryptographic APIs [22] and their programming tricks [23]. Finally, there are software security books covering crypto programming [24]–[27] and giving design advice [6][28][29]. A few security resources also give practical advice for secure crypto programming and design: *(i)* the top-10 security risks [30] and testing guide [31]; *(ii)* the top-10 secure design flaws [32]; *(iii)* the practices for secure software development [33]; and *(iv)* the top-25 most dangerous programming errors [34].

A. Cryptography and the SSDL

In general, software security recognizes cryptography's importance and complexity. However, cryptography stays at low-level design, as a feature or mitigation control used in simple tactics to achieve security goals. In fact, for

978-1-5090-4388-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSSA.2016.12

practitioners [5] and researchers [6], cryptographic security is a feature to be added rather than a property to be enforced. Thus, SSDL methods in general have no specific activities for it.

The reality nowadays is often that secure systems have to perform complex functionality while avoiding attacks. Sadly, common cryptographic security only provides usual use cases for securing data at rest or in transit, consisting of data encryption, integrity checks, user authentication, non repudiation, and key management. These tasks are simple scenarios for the broad concept of cryptographic feature, including not only cryptographic algorithms and protocols, but also software libraries and frameworks. The software security community agrees in not investing effort to implement proprietary cryptography. Instead, crypto features should be obtained from trusted infrastructures and service providers. We believe that this posture oversimplifies cryptography's role in software security.

It is believed [6] that the development of cryptographic software is better served by a waterfall-like SSDL, because of the high predictability and ease of evidence-gathering provided by this kind of lifecycle. This view reminds us of the Common Criteria for Information Technology Security Evaluation [35] and the Cryptographic Module Validation Program [36]. We believe that this is an extreme view that should not be adopted as is, but be adapted to fit modern software development performed by common developers.

Also, historically, programmers have not been trained to consider the capabilities of an adversary. This results in protections against problems familiar to programmers, but that are still easy for attackers to subvert [26]. We believe that this is one of the causes cryptography misuses have not been directly approached into the SSDL. Security tools are often adopted, but they still require the expertise of experienced security practitioners [37]. We add expertise of cryptography architects as well, because developers should understand it in order to avoid misusing it [29].

B. *Cryptography into the SSDL*

This section synthesizes cryptography activities into SSDL phases, according to SWEBOK [38]. A comparison of DSCS to main SSDL methods is shown in TABLE I.

1) *Cryptography into Software Requirements Security*

In software requirements security, cryptographic needs are identified and obtained from various sources, including security policies and regulations, business needs, non-functional requirements, and predefined checklists. Also, cryptographic features have been derived from the basic security goals of confidentiality, integrity, availability, authenticity, non-repudiation, and privacy [37][39]. The early importance given to cryptography during gathering and elicitation of security requirements generally depends on the interest of stakeholders in the subject [40].

In order to avoid implicit requirements, developers usually adopt simple ways to specify cryptography requirements, often using checklists of controls against known vulnerabilities, such as the OWASP top-10 [30].

2) *Cryptography into Software Design Security*

Policies and the target environment drive the design of security architecture [41]. Secure design principles enforce properties in order to avoid design flaws and secure profiles add predefined controls against known threats [41]. The security architecture enforces the design goal of eliminating weak encryption by avoiding known design pitfalls [32], such as: use of proprietary algorithms or implementations, misuse of libraries and algorithms, poor key management, bad randomness, decentralized crypto infrastructures, and unsupported crypto evolution.

Only recently, secure design initiatives emerged as an industry-led effort and began to consider cryptography misuse as a secure design issue. For instance, the IEEE CYBSI [32] considers the design pitfalls for cryptography mentioned above. Also, Safecode [33] considers that the elimination of weak encryption from software is the most important design issue related to cryptography today.

3) *Cryptography into Software Construction Security*

Coding guidelines have been adopted to integrate cryptographic features to source code [42][26]. In general, developers avoid or prevent many security issues (e.g., timing attacks and weak keys) just by using good cryptographic libraries [6]. Also, cryptographic agility allows for rapid adaptation for new cryptography [41].

We distinguish secure coding of crypto software and coding secure crypto software. The former uses general secure coding techniques during the coding of crypto software. The latter embraces specific secure coding techniques and programming tricks to better defend crypto software against particular misuses and bad construction of cryptographic techniques.

A general concern is to avoid code-level vulnerabilities when using frameworks and libraries. In general, static analysis is the most cost-effective way to find known vulnerabilities [26] and manual inspection is usually adopted for small portions of code [5]. This should apply to the detection of cryptography misuses. However, in practice, secure coding is limited to standards enforced by language features, security frameworks, and simple tools.

4) *Cryptography into Software Testing Security*

Crypto software testing usually looks for weak cryptography issues, including functional tests of features and penetration tests for known vulnerabilities. These tests have been applied to simple functionality that uses cryptographic features, as well as to cryptographic packages encapsulated by high-level frameworks, such as Transport Layer Security's (TLS) libraries [12][13].

Software security organizations are concerned with weak encryption in standardized protocols. For instance, the OWASP testing guide [31] gives test procedures for three types of issues on weak encryption: unencrypted HTTP channels, insufficient TLS protection, and padding oracles. In general [24]-[28], the software industry favors operational aspects of security, with interest in penetration tests and security assessments, before releasing software to production or distribution. Also, in operation, specialized penetration tests are periodically applied against protocols and features, showing a relatively high cost for correction.

IV. THE PROPOSED METHODOLOGY

As far as we know, there is no documented approach to explicitly include cryptography into SSDL and that could be adopted by modern software factories. However, we have observed the frequent adoption of common practices by researchers and practitioners. This observation allows for an informal documentation of the methodology for Development of Secure Cryptographic Software (DSCS), which is detailed in the next subsections.

A. *Steps for DSCS*

DSCS comprises a set of steps related to SSDL phases [38] and an extra phase for deployment. The concept of assurance gates is used to informally capture the idea of traceability among steps and of returning to previous steps when assurance requirements are not achieved. Figure 1(a) captures these steps along with SSDL phases. In the figure, orange rectangles are input information, red rectangles are wisdom captured from crypto experts, green rectangles represent knowledge base, blue rectangles are expected software deliverables, and yellow rectangles are assurance gates. The following paragraphs describe the steps.

Step 1: Crypto software requirements security. Security policies, regulations, business needs, non-functional requirements, and predefined checklists determine which security goals must be accomplished. Security goals (e.g., anonymity, confidentiality, integrity, non-repudiation, authenticity, etc.) are then mapped to predefined cryptographic features. Those mappings can follow both common user stories and developer's coding tasks, facilitating the detection and avoidance of crypto misuses. The first assurance gate validates the transition from security requirements to cryptographic features. This step is usually supported by software security practitioners or security architects. Crypto experts support the specification of new user stories (use cases) as well as the related programming tasks and crypto features.

Step 2: Crypto software design security. Cryptographic features, design goals, and predefined control types contribute to define a security architecture, which contains cryptographic services and technologies. In this step, cryptographic design patterns can provide proven solutions to common use cases and coding tasks traditionally associated to cryptography. The second assurance gate validates the transition from cryptographic features and control types to actual cryptographic services and technologies. This step is usually supported by security architects or software security practitioners. Cryptography experts may provide timely advice.

Step 3: Crypto software construction security. This step consists in programming secure crypto software. Code is written to integrate crypto controls and features into software functionality. The source code must follow conventions of crypto APIs and adopts standard implementations of algorithms and protocols, which are offered by frameworks and libraries. The third assurance gate validates the transition from crypto features and controls to APIs, libraries, and frameworks, and helps to avoid crypto misuse. In this step, the work of software security practitioners receives most support from experts in applied cryptography in order to avoid crypto misuses.

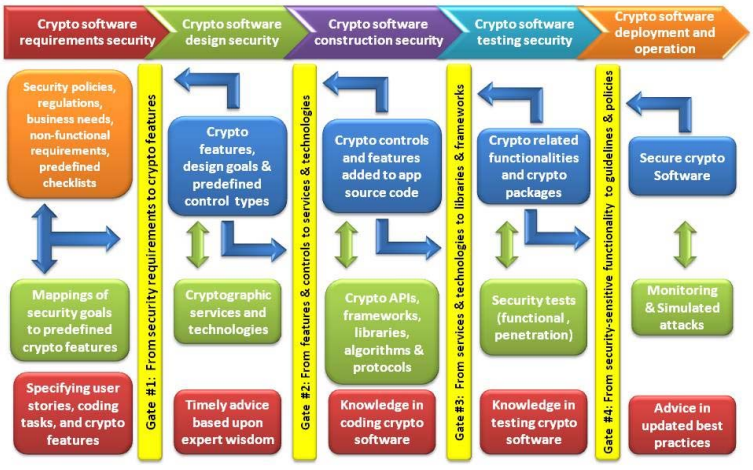
Step 4: Crypto software testing security. Cryptography-related functionality and crypto packages are submitted to security tests of two types: functional security tests (supported by security-inspired test cases) and penetration tests, supported by attack scenarios and threats. The fourth assurance gate validates the transition to deployment and operations by assuring compliance to crypto guidelines, policies, and requirements. In this step, the work of software security practitioners is supported by experts in cracking crypto software, usually by exploiting remaining crypto misuses and other implementation bugs.

Step 5: Crypto software deployment and operation. In this step, secure software is continuously monitored and periodically tested for violations of policies and guidelines, as well as for discovery of new vulnerabilities. This step is usually performed by a software security practitioner. Continuous attention should be given to the fast and constant evolution of cryptographic technology in terms of new standardized algorithms, updated best practices, and adoption of longer key lengths.

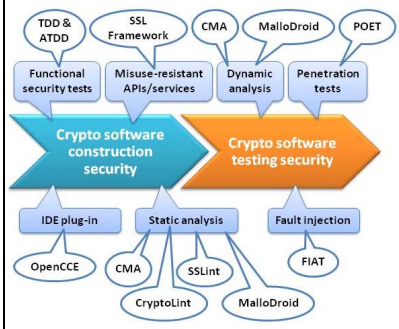
The above steps and activities are quite straightforward and capture the common practices we have seen so far. They may not be the best choices made by practitioners, but the possible ones due to several constraints. For instance, it is quite common to involve cryptography experts only in later steps of SSDL (e.g., coding and testing). Also, the above steps do not explicitly handle software architectures or tool support. These, however, are essential items for modern software development.

B. *Layers for Crypto Software in DSCS*

Nowadays, crypto software might not only follow the



(a) Steps for DSCS along with SSDL phases.



(b) A toolkit for secure coding and verification of crypto software.

Figure 1. The methodology for Development of Secure Cryptographic Software (DSCS).

prescription of cryptography experts, but also be structured in layers of abstraction according to software engineering best practices. Micro architectures for crypto software have already been proposed in a previous work [43]. This section goes further in discussing high-level abstractions for crypto features, proposing layers for crypto software. We give a short description for each of these layers:

1. *The user interaction layer*. This layer has to be able to promote the proper use of cryptography, inhibiting the misuse of cryptographic features by final users when handling security-sensitive functionality;
2. *The business logic layer*. This layer should be able to properly orchestrate crypto services and components, with the adequate design abstractions expected by developers when securing sensitive business goals;
3. *APIs and frameworks layer*. This layer must be able to provide access to cryptographic implementations in standardized and decoupled ways, so that both the replacement and the exchange of implementations are easily achieved. Also, APIs, libraries, and frameworks should not disclose sensitive information through unauthorized side channels;
4. *Algorithm and protocol implementation layer*. In this layer, cryptographic implementations of algorithms and protocols should be robust against various failures (e.g. hardware and memory failures), secure against various attacks (timing, side-channel leakages, etc.), efficient in energy (low power consumption) and computational resources (CPU cycles, memory, etc.), and compact for use in restricted environments;
5. *Development support and infrastructure layer*. In this layer, programming languages, component libraries, compilers, obfuscators, and even operating systems should be able to capture the programmer's intent, preserving security decisions, and not canceling protections when translating source or binary code of cryptographic implementations to machine code.

All layers of crypto software must be reliable and useful in contributing to the security of the software as a whole, working together to promote the secure use of cryptography. In addition, we recognize the importance of cryptography's theoretical security. However, in this work, the mathematical security of algorithms and protocols is an unspoken assumption, taken for granted.

Finally, layers can be associated to DSCS steps, in specific instantiations. For instance, the user interaction layer may relate to requirements elicitation and business modeling (software requirements security); the business logic layer may relate to design and architecture (software design security); the API and frameworks layer may relate to secure coding and construction (software construction security); and security of crypto components is related to verification and validation against predefined requirements and new threats (software testing security).

C. Analysis of Tool Support for DSCS

The development of secure crypto software still lacks professional tool support for most issues, which are currently subject of academic research. The development of proprietary cryptography (proprietary algorithms or homemade implementations) by ordinary programmers is considered bad practice, so we omitted in this text all those tools for secure programming of crypto libraries (i.e., tools applied below crypto APIs), including domain-specific programming languages, secure compilation, and automated code generation. A previous work [44] surveys tools for secure coding and testing of crypto software. Here in this work, we focus on tools for development of cryptography-enabled functionality with established and standardized algorithms, as well as trusted libraries and frameworks (i.e., tools applied above crypto APIs). A toolkit for DSCS, illustrated in Figure 1(b), may include tools for both secure programming and verification.

For crypto software construction, SSL/TLS frameworks [45] offer high-level functionality and configurable services, instead of primitive functions, enhancing usability and avoiding unintentional mistakes. Also, specialized tools, such as OpenCCE [46], can be integrated to IDEs and guide developers through selection and use of relevant crypto components, automatically generating code with suitable API calls, and analyzing the final code, avoiding mistakes by non-experts.

For static analysis, current coding standards [30][34] do offer simple rules for cryptography misuse that can be automated by simple tools. However, sophisticated issues cannot be detected by ordinary tools and have only been addressed by prototypes. For instance, CryptoLint [11] and Cryptography Misuse Analyzer (CMA) [14] are both static analysis tools for Android apps that identify predefined sets of misuses and vulnerabilities from API calls. Also, both SSLint [47] and MalloDroid [13] are static analysis tools for detecting incorrect use of SSL/TLS APIs and improper certificate validation, detecting potential man-in-the-middle attacks.

For crypto software testing security, tests for SSL have been used for detection of HTTPS misconfigurations in web apps [31]. Also, MalloDroid [13] can be used to dynamically detect SSL vulnerabilities against Android apps. The Padding Oracle Exploitation Tool (POET) [48] automatically finds and exploits this type of side channel. Also, Fault Injection Attack Tool (FIAT) [49] can inject malicious faults into cryptographic devices.

Finally, experience shows that tools are incomplete, not overlapping, and buggy. Thus, various tools should be combined to obtain diversity and redundancy, promoting fault tolerance to DSCS. The incompleteness and absence of tools in testing is more evident than in construction. In practice, household tool development and customization, as well as manual testing skills, have been favored.

D. Comparison of DSCS to Other SSDL Approaches

TABLE I qualitatively compares DSCS to main SSDL approaches [5][28][51] and clarifies the differences between our methodology and other methods. First, the technology-driven approach [5] favors security in primary software deliverables, such as source code, architectural design, tests, and executables. Second, the process-driven approach [28] emphasizes threat modeling, well-defined processes, and vulnerability management. Third, the risk-management approach [41][51] overstresses risk analysis and assessment, as well as security management.

TABLE I shows that other methods are too broad to be useful for development of modern crypto software, but they can still be extended by DSCS, because it highlights

usually neglected crypto issues, empowering developers with proper ways to build crypto software. For instance, practitioners sometimes show a bias to process-driven approaches, preferring the predictability of a waterfall-like SSDL driven by known requirements, resulting in software able to pass well-defined evaluation criteria.

Along the same line, other practitioners show a bias towards the technology-driven approach, arguing that cryptographic features are mostly well-defined controls that will be put in place anyway, with no need for deep threat modeling or traceable processes. For them, coding guidelines supported by tools and a good (secure enough) architecture are sufficient to securely apply cryptography. In general, there is no particular interest in applying fine-grained risk analysis deeply into daily activities for crypto software, leaving risk analysis at the business level.

TABLE I shows that SSDL methods have no specific activities for cryptography. However, the development of crypto software suffers great influence from the supporting SSDL, determining the way cryptography is approached. For instance, modern software development frequently adopts highly adaptive lifecycles [50], characterized by progressive specification of requirements based on short iterative development cycles. In this way, a technology-driven DSCS is a better fit for modern crypto software built within highly adaptive lifecycles, where processes are not rigid paths and risk analysis has to be quick enough to not slow down the team, reducing risk gradually over time by the evolution of understanding.

In our experience, a technology-driven DSCS can be adopted bottom-up and conducted by technical staff (supported by executives). For instance, a lower manager can enhance the software security capabilities of his staff by integrating into the SSDL a well-crafted set of tools and techniques for crypto software security, adopting Acceptance Test-Driven Development (ATDD) during the construction of cryptographic services [52].

E. Examples of Practical Inspiring Cases for DSCS

DSCS emerged as a response to practical needs when proposing crypto design patterns [43], investigating crypto misuses [53], building secure mobile apps [1]-[4], and applying ATDD for crypto services [52].

Several inspiring cases gave us the required perspective to generalize a working methodology able to handle unusual use cases for cryptography. For instance, during the development of an app for secure, end-to-end instant messages [1], the key agreement protocol had to be blended to the chat service, the transport of group-chat keys had to be blended to chat rooms, and perfect forward secrecy was preserved despite the storage of chat history.

In a second development case, an app for authenticated encryption of SMS [2], the combination of encryption and digital signatures was a common source of design flaws, the transport of shared keys was masked as invitations to be a contact, and key update was blended to user notifications and app updates. In a third case, an app for encrypted file system with secure deletion of files [3], secure deletion was obtained by purging crypto keys from storage, and copies of a file could not be deterministically encrypted to be equal. These development cases were part of a framework for cryptographically secure technologies on mobile devices [4].

Finally, we noted that, without DSCS, developers are likely to avoid entering the hardest cryptography issues, minimizing them to high-level, theoretic knowledge and punctual advice from experts. Many times, we have seen practitioners complaining about cryptologists' tendency to foster theory, neglecting practical issues, such as crypto misuse cases and implementation bugs. This behavior led practitioners to react negatively to cryptography issues, sometimes adopting high-level risk analysis as the only assurance control available and limiting activities to vulnerability management during operations and software patching after system compromise.

TABLE I. DSCS COMPARED TO OTHER SSDL APPROACHES.

	Technology-driven SSDL [5] (a.k.a. McGraw's Touchpoints)	**Process-driven SSDL [28] (a.k.a. Microsoft SDL)**	**Risk-driven SSDL [51] (a.k.a. Risk management)**	**Development of Secure Cryptographic Software (DSCS)**
Remarkable feature	Associated to main software deliverables: source code, architecture, tests, and executables	Threat modeling, well-defined development processes, and vulnerability management	Security is a business issue. Risk-analysis provides evidence of due diligence for security management	Highlights usually neglected crypto issues, providing better ways to build crypto software in software factories
SSDL enforcement	Enforces process enhancement by hardening SDLC processes with security tools and techniques	Integrating security mindset and processes into SDLC is the reliable way to ensure software security	Project and risk management occur together. Preliminary risk assessment gives initial security requirements	Fits any SSDL, specific activities and roles applied to adaptive lifecycles and technology-driven SSDLs
Software Requirements Security	Security checklists, threat modeling, and misuse cases	Security requirements, awareness, and training complement threat modeling and direct architecture	Assets and threats identified in requirements. Risk acceptance criteria for prioritizing mitigations.	Crypto use cases and coding tasks, feature checklists, catalogs of crypto misuses based on expert knowledge
Software Design Security	UML for secure design, attack patterns, security patterns, and architectural analysis for security	Design reviews for attack surface reduction, threat modeling, prioritized controls, and security architecture	Threat modeling binds design and risk analysis, attack surface reduced by threat modeling and security assessment of architecture	Layers for crypto software, catalogs of crypto misuses and design flaws, code samples, reusable libraries and frameworks
Software Construction Security	Static analysis tools, code reviews, security guidelines, and coding standards for secure coding	Code analysis and reviews mitigate vulnerabilities, version control tracks changes and enables rollback	Common practices to mitigate risk: code reviews and inspections, unit testing, and static analysis	Coding standards and guidelines, reviews for crypto misuses, code samples for libs and frameworks
Software Testing Security	Test cases for security requirements, risk-based test cases for common vulnerabilities, and pentesting	Security tests verify features for nonfunctional requirements, pentests for common vulnerabilities	Simulated attacks and penetration tests based on risk assessment and prioritization	Tests for crypto misuses on crypto use cases and tasks, pentests for common crypto vulnerabilities
Deployment & Operations Security	Final security review before release and plans to incident response	Final risk assessment, before release, gives confidence to accept release of software	Final risk assessment needed to accept risk and securely authorize deployment of software release	Monitoring and assessment, tests for new vulnerabilities, watch for crypto evolution and security updates

V. Concluding Remarks

Over the years, cryptography has not only been misunderstood and misused by software developers, but also underestimated by SSDL methods. DSCS bridges the gap between modern cryptographic software and actual development methods for secure software. We do believe that DSCS, in its current stage of maturity, is a step forward in providing better ways to build crypto software in common software factories. Future works include usability (behavioral) experiments and field tests concerning developers' acceptability of DSCS. Also, we see tool development as fertile area of research, especially for design and architecture of secure crypto software.

Acknowledgment

Alexandre Braga thanks CNPq and Intel for the financial support. Ricardo Dahab thanks FAPESP, CNPq, CAPES, and Intel for partially supporting this work.

References

[1] A. Braga and D. Schwab, "Design Issues in the Construction of a Cryptographically Secure Instant Message Service for Android Smartphones," in the 8th SECURWARE, 2014, pp. 7–13.

[2] A. Braga, R. Zanco Neto, A. Vannucci, and R. Hiramatsu, "Implementation Issues in the Construction of an Application Framework for Secure SMS Messages on Android Smartphones," in the 9th SECURWARE, 2015, pp. 67–73.

[3] A. Braga and A. Colito, "Adding Secure Deletion to an Encrypted File System on Android Smartphones," in the 8th SECURWARE, 2014, pp. 106–110.

[4] A. Braga, "Integrated Technologies for Communication Security on Mobile Devices," in 3rd Mobility Conf., 2013,pp. 47–51.

[5] G. McGraw, Software Security: Building Security in. 2006.

[6] R. Anderson, "Security engineering," 2008.

[7] B. Schneier, "Cryptographic design vulnerabilities," Computer, Sept, pp. 29–33, 1998.

[8] P. Gutmann, "Lessons Learned in Implementing and Deploying Crypto Software," Usenix Security Symposium, 2002.

[9] R. Anderson, "Why Cryptosystems Fail," in Proc. of the 1st ACM Conf. on Computer and Commun. Security, 1993, pp. 215–227.

[10] B. Schneier, "Designing Encryption Algorithms for Real People," Proc. of Workshop on New Security Paradigms., pp. 98–101, 1994.

[11] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," ACM Conf. on Comp. & Comm. Sec., pp. 73–84, 2013.

[12] M. Georgiev, S. Iyengar, and S. Jana, "The most dangerous code in the world: validating SSL certificates in non-browser software," in Proc of ACM Conf. on Comp. and Comm. Sec., 2012, pp. 38–49.

[13] S. Fahl, M. Harbach, and T. Muders, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," in ACM Conf. on Comp. and Comm. Sec., 2012, pp. 50–61.

[14] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications," in IEEE 12th Intl. Conf. on Dependable, Autonomic and Secure Computing, 2014, pp. 75–80.

[15] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why Does Cryptographic Software Fail?: A Case Study and Open Problems," in 5th Asia-Pacific Workshop on Systems, 2014, pp. 7:1–7:7.

[16] "The Heartbleed Bug." [Online]. Available: http://heartbleed.com/.

[17] "Apple's SSL/TLS 'Goto fail' bug." [Online]. Available: www.imperialviolet.org/2014/02/22/applebug.html.

[18] T. Jager and J. Somorovsky, "How to break XML encryption," Proc. of 18th ACM Conf. on Comp. and Comm. Sec, p. 413, 2011.

[19] T. Duong and J. Rizzo, "Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET," IEEE Symp. on Security and Privacy, pp. 481–489, May 2011.

[20] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, Handbook of applied cryptography. CRC press, 1996.

[21] N. Ferguson, B. Schneier, and T. Kohno, Cryptography Engineering: Design Principles and Practical Applications, 2011.

[22] J. B. Knudsen, Java Cryptography. Oreilly, 1998.

[23] D. Hook, Beginning cryptography with Java. 2005.

[24] J. Viega and G. McGraw, Building Secure Software: How to Avoid Security Problems the Right Way. 2001.

[25] M. Howard and D. LeBlanc, Writing secure code. 2003.

[26] B.Chess and J.West, Secure programming w/ static analysis. 2007.

[27] M. Howard, D. LeBlanc, and J. Viega, 24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them. 2009.

[28] M.Howard and S.Lipner, Security Development Lifecycle. 2006.

[29] A. Shostack, Threat modeling: Designing for security. 2014.

[30] "OWASP Top Ten Project," OWASP, 2013. [Online]. Available: https://www.owasp.org/index.php/Top_10.

[31] "OWASP Testing Project v4," OWASP, 2015. [Online]. Available: https://www.owasp.org/index.php/OWASP_Testing_Project.

[32] "Avoiding The Top 10 Software Security Design Flaws," IEEE Cybersecurity Initiative (CYBSI), 2014. [Online]. Available: http://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf.

[33] "Fundamental Practices for Secure Software Development," Safecode, 2011. [Online]. Available: http://www.safecode.org/wp-content/uploads/2014/09/SAFECode_Dev_Practices0211.pdf.

[34] "TOP 25 Most Dangerous Software Errors," SANS/CWE. [Online]. Available: www.sans.org/top25-software-errors.

[35] "Common Criteria for Information Technology Security Evaluation - Part 1: Introduction and general model." 2012.

[36] NIST, "Cryptographic Module Validation Program (CMVP)." [Online]. Available: csrc.nist.gov/groups/STM/cmvp/index.html.

[37] J. Ransome and A. Misra, Core Software Security. 2013.

[38] P. Bourque and R. Fairley, Eds., Guide to the Software Engineering Body of Knowledge (SWEBOK), V3. 2014.

[39] M. Merkow and L. Raghavan, Secure and Resilient Software Development. 2010.

[40] T. Richardson and C. Thies, Secure Software Design. 2012.

[41] M. Paul, Official (ISC)2 Guide to the CSSLP. 2011.

[42] J. Allen, S. Barnum, R. Ellison, G. McGraw, and N. Mead, Software Security Engineering: A Guide for Proj. Managers. 2008.

[43] A. Braga, C. Rubira, and R. Dahab, "Tropyc: A pattern language for cryptographic object-oriented software," in PLoP, 1998.

[44] A. Braga and R. Dahab, "A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software," in proc. of XV SBSeg, 2015, pp. 30–43.

[45] S. Fahl, M. Harbach, and H. Perl, "Rethinking SSL development in an appified world," Proc. of ACM Conf. on Comp. & Comm. Sec., 2013, pp. 49–60.

[46] S. Arzt, S. Nadi, K. Ali, E. Bodden, S. Erdweg, and M. Mezini, "Towards Secure Integration of Cryptographic Software," in ACM Intl. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software, 2015, pp. 1–13.

[47] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting SSL usage in applications with SSLint," in IEEE Symp. on Sec. and Privacy, 2015, pp. 519–534.

[48] J. Rizzo and T. Duong, "Practical padding oracle attacks," Proc. of the 4th USENIX conf. on Offensive technologies, pp. 1–9, 2010.

[49] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices," in Proc. of the IEEE, vol. 100, no. 11, pp. 3056–3076.

[50] PMI, Software Extension to the PMBOK® Guide, 5th ed. 2013.

[51] S. Harris, CISSP All-in-One Exam Guide, 6th Edition. 2012.

[52] A. Braga and D. Schwab, "The Use of Acceptance Test-Driven Development to Improve Reliability in the Construction of Cryptographic Software," in the 9th SECURWARE, 2015.

[53] A. Braga and R. Dahab, "Mining Cryptography Misuse in Online Forums," in 2nd IEEE Intl. Workshop on Human and Social Aspect of Software Quality, 2016.

2.3.2 Understanding the Field of Cryptographic Software Security

This publication is entitled *"Understanding the Field of Cryptographic Software Security"* and was submitted to publication, in November 2017, to the Journal of Information and Software Technology, maintained by the publisher Elsevier.

Understanding the Field of Cryptographic Software Security

Alexandre Braga[a], Ricardo Dahab[a]

[a]*State University of Campinas (UNICAMP), São Paulo, Brazil*

Abstract

Context. Cryptography is an essential part of software security and one of the most misunderstood technologies in the development of secure systems. Its correct use is full of design decisions and coding pitfalls that confuse the ordinary developer. The investigation of real-world issues associated to cryptography misuse is gaining momentum, as many experts believe now that cryptographic software security is not only a matter of secure implementation of cryptographic algorithms, but also includes correct usage of APIs, adequate system design, and proper verification tools to avoid cryptography misuse.

Objective. This paper investigates the role of cryptography in software security, preparing the stage for an emerging field of study concerned with the development of secure cryptographic software.

Method. This paper analyses five complementary views for the development of cryptographic software. Two mapping studies contribute to understand the role of cryptography in software security and propose a classification of cryptography misuse for software security. Two empirical studies regarding cryptography misuse in online communities and one experimental evaluation of static analysis tools for cryptography validate the proposed classification. Then, a working methodology is customized to better fit cryptography in software security.

Results. The paper assembles a body of knowledge for this new field of study named **Cryptographic Software Security**, which is supported by a validated classification of cryptography misuse and a working methodology for developing secure cryptographic software.

Conclusion. The better the understanding regarding the role of cryptography in software security, the most effective can be the tools and techniques for preventing, detecting, and mitigating cryptography misuse in software systems. This work provides a validated background to appropriately position research activities.

Keywords: Software development, cryptography, software security, mapping study, cryptographic software security

1. Introduction

When cryptography was restricted to a relatively small number of application types, cryptography insecurity was mainly associated to flaws in the underlying mathematical assumptions of cryptographic algorithms or to broken implementations of those algorithms. Nowadays, many popular cryptographic software are developed by ordinary developers without expert help and massive deployments of cryptosystems transparently manage millions of keys on behalf of their users. In this context, the most frequent security issue related to cryptography is its misuse.

Email addresses: ambraga@cpqd.com.br (Alexandre Braga), rdahab@ic.unicamp.br (Ricardo Dahab)

However, in spite of the growing popularity of cryptographic services, historically, practitioners have always reserved little attention to cryptography misuse, compared to other sources of vulnerabilities. This situation contributed to the widespread misuse of cryptography by a multitude of ordinary software developers.

Over the years, software security evolved from lists of good advice (supported by vulnerability descriptions and countermeasures) to a well-defined body of knowledge with recognized methods, tools, and techniques. Nevertheless, in general, the software security does not adequately address the issues of cryptographic software security, and the methods for development of secure software have no specific activities for cryptography. In fact, cryptography

has been considered only a security feature added to software during coding, with no specific support from development processes.

Cryptographic Software Security is a broad subject that involves at least three distinct communities, that have drifted apart from each other: (applied) cryptologists, (software) security experts, and software developers. Cryptologists are used to focus on the mathematical security of cryptography and secure implementation of algorithms. Security experts frequently only react to exposed vulnerabilities in the cryptographic infrastructure, after software compromises. Software developers usually take for granted the security of cryptographic libraries, overlooking the trick details of cryptography usage that can lead to system vulnerabilities. This state of affairs contributed to an adversarial posture among communities.

For instance, many times, we saw practitioners complaining about cryptologists' tendency to foster theory and neglect practical issues associated to cryptography misuse. Also, many times, practitioners adopt high-level risk analysis as the only assurance control available, limiting activities to vulnerability management during operations and software patching after system compromise.

Without an ordered way to mitigate cryptography misuse in early stages of software development, developers are likely to overlook the hardest cryptography issues, underestimating their complexity, and minimizing them to high-level, theoretic knowledge and punctual advice from experts. Therefore, there is a need for methods to build secure cryptographic software, in such a way that cryptographic security could be easily blended to application's functionality.

The objective of this text is to set the stage for a new field of study named Cryptographic Software Security. To accomplish this goal, we correlate five facets of this emerging field of study (a sub field of software security) concerned with the development of secure cryptographic software and refine a methodology to better fit cryptography into secure software development.

In this text, we detail two mapping studies. The first explores the broad field of software security, from the view given by textbooks, in order to understand the state-of-practice regarding how cryptography is approached by software security. The second mapping study complements the first and explores the cryptography knowledge found in software security textbooks, building a classification of cryptography misuse for software security.

We also discuss two empirical studies and one experimental evaluation, which were used to validate the proposed classification of cryptography misuse. The empirical studies analyzed the occurrence of cryptography misuse in coding communities, studying how specific misuse categories relate to each other, and determining whether users of these communities improve their skills in cryptography with time. The experimental evaluation is a benchmark of static code analysis tools that measures the capabilities of free tools in detecting cryptography misuse related to coding, design, or architectural issues. Finally, we applied our findings to customize a working methodology for development of secure cryptographic software. This work sits in the intersection of three domains (e.g., applied cryptography, software engineering, and software security) sharing a common body of knowledge.

The main contributions of this investigation are the following. First, two mapping studies of how software security approaches applied cryptography. Second, a classification of cryptography misuse for software security. Third, the joint analysis of two empirical studies and one experimental evaluation to validate the proposed classification. Fourth, the refinement of a working methodology for development of secure cryptographic software. Fifth, the characterization of a new field of study concerned with cryptographic software security.

The remaining parts of this text are organized as follows. Section 2 gives necessary concepts and examples of cryptographic software. Section 3 exposes how software security supports cryptographic software. Section 4 details how cryptography misuse emerges from software security issues and proposes a classification for them. Section 5 describes how developers misuse cryptography in online communities. Section 6 describes how static analysis tools detect cryptography misuse. Section 7 customizes a methodology for development of secure cryptography software. Section 8 discusses results and Section 9 makes concluding remarks.

2. Background

This section gives concepts and examples of cryptographic software, which are necessary for the understanding of the viewpoint proposed in this text.

2.1. Concepts

The ordinary way to design, build and test software is usually named Software Development Life Cycle (SDLC). Secure Software Engineering (SSE), also named **Software Security**, is the process of designing, building, and testing software so that it becomes secure [1]. SSE aims at avoiding vulnerabilities in software by considering security aspects since from the very beginning and throughout the SDLC. SSE includes processes for Secure Software Development Life Cycles (SSDLC). The difference between SSDLC and SDLC is that the former focuses on security and privacy, while the latter is concerned with improving the quality in general, usually with no specific interest for security.

Cryptographic software (crypto software, for short) is software that preserves major security goals (namely, confidentiality, integrity, authenticity, and non repudiation) transparently blended into functionality, by using cryptographic methods (services or mechanisms) available through reusable cryptographic infrastructures. In this context, **applied cryptography** is the use of cryptographic infrastructures (e.g., packages, libraries, frameworks, modules, and APIs) to build cryptographic security within SSDLC efforts.

Cryptography misuse (crypto misuse, for short) is a programming bad practice frequently found in cryptographic software, leading to exploitable vulnerabilities, and introduced by developers during coding tasks associated to use cases enabled by cryptography. We do not simply name these misuses as vulnerabilities because, in many cases, they are design flaws and insecure architectural choices with code-level consequences. Crypto misuse is not related to implementation of cryptographic algorithms. Instead, crypto misuses emerge when ordinary developers use cryptographic infrastructures in their daily coding activities during the development of cryptographic software.

Crypto misuse is not simply a program fault in the code, either. It is more like a program anomaly. According to Sommerville [2], (coding) anomalies are often a result of programming errors or omissions that highlight things that could go wrong when the program is executed. However, anomalies are not necessarily program faults; they may be deliberate constructs introduced by the programmer.

2.2. Examples of cryptographic software

This section describes three instances of modern cryptographic software that go far beyond traditional use cases for cryptography (e.g., SSL security, file encryption, or password protection), and that require cryptographic protocols to be blended to application's functionality: *(i)* secure, end-to-end instant messages [3] (in subsection 2.2.1); *(ii)* authenticated encryption for SMS [4] (in subsection 2.2.2), and *(iiii)* encrypted file systems with secure deletion [5] (in subsection 2.2.3). These development cases were part of broader framework for cryptographic security on mobile devices [6].

2.2.1. Cryptographically Secure Instant Messages

This study case covered the design and implementation issues in the construction of a cryptographically secure Instant Message (IM) application for Android and the underlying cryptographic library that supported it [3]. Cryptographic services were crafted to adequately fit to a secure IM service in a way that was transparent to users, without sacrificing security. A well-defined architecture allowed the selection and use of non-standard cryptography through a standard API.

During this development case, the key-agreement protocol had to be transparently blended to the chat service, the transport of group-chat keys had to be blended to broad metaphor of chat rooms, and forward secrecy was preserved despite the storage of chat history.

This mobile app had three main use cases for cryptography: secure communication (SC) with key agreement, encrypting data at rest (EDR) for secure storage of chat conversations, and authentication and validation of data (AVD) for an authenticated key agreement in secure communication (SC). In its architecture, this mobile app needed a certification authority (CA) and public-key infrastructures (PKI) as external services. A complex architecture for key management blended to application functionality amplified crypto misuses from public-key cryptography (PKC) issues in key agreement and improper certificate validations (ICV).

This development case needed expert support during coding, design reviews, architecture (security) assessment, and security testing. Other concerns needing expert attention were the detection of insecure parameters and the avoidance of insecure settings in Android. Thus, a cryptography expert was needed from requirements to testing.

2.2.2. Authenticated Encryption for SMS

An application framework for SMS security provided secrecy, integrity, authentication, and non re-

pudiation for Short Message Service (SMS) on mobile devices [4]. The framework integrated authenticated encryption and short digital signatures to management services for keys and certificates, hiding from final users all details concerning certificate and key management.

Distinct trade-offs between security and message length resulted in different levels of security: secrecy only, secrecy with message authentication, and secrecy with origin authentication and non repudiation. The application framework used short signatures (a non-standard cryptographic function) for authenticating the origin of SMS messages, packing in a single, 140-byte array the information needed to authenticate the origin of messages, while still allowing a useful length of text for the message.

In this development case, the combination of encryption and digital signatures was a common source of early-detectable design flaws, the transport of shared keys was masked as invitations to become a (phone) contact, and the update of keys was blended to notifications and app updates, to match the usage style of mobile apps. Authentication and validation of data (AVD) and secure communication (SC) were the two main use cases. Its architecture had to consider distribution of public keys for the signature cryptosystem, as well as integration to external services for CA. A preliminary security assessment showed that this app had chance to suffer from misuses related to IV management (IVM) and distribution of secret keys (affected by poor key management - PKM), program design flaws (PDF) due to combination of encryption and digital signatures, improper certificate validation (ICV), and the common issues of weak cryptography (WC) and coding and implementation bugs (CIB).

This development case showed a relatively complex architecture for key management blended to app functionality. Like the previous example, this development needed expert support during coding, design review, and architecture assessment. Security testing was applied for detection and avoidance of insecure parameters for digital signatures, iv management, and improper certificate validation. Again, an expert in cryptography was needed from requirements to testing.

2.2.3. Encrypted File System with Secure Deletion

This study case implemented two user-level approaches to perform secure deletion of files [5]. One working on secure deletion of encrypted files, and other handling deletion assurance for ordinary (unencrypted) files. Copies of files were randomly encrypted to not be equal. Also, secure deletion of encrypted files was obtained by purging crypto keys from storage, and was fully integrated to an encrypted file system, being transparent to users. Secure deletion of ordinary files was fulfilled by an autonomous service activated under the discretion of the user.

This app had three main use cases for cryptography: authentication and validation of data (AVD), encryption of data at rest (EDR), and the unusual secure file deletion. Its development was likely to be affected by crypto misuses related to weak cryptography (WC), IV and nonce management (IVM) issues, and program design flaws (PDF), when combining encryption and authentication. Also, the first use case had to avoid side channels in verification of file integrity (a kind of design flaw), the second use case had to avoid deterministic encryption (a kind of weak cryptography), and the third use case had to avoid unauthorized access to undeleted data due to flawed management of keys.

Because of known use cases associated to a reusable component (an encrypted file system), the development needed expert support during coding, design review, and architecture assessment. Thus, needing expert help from design to testing.

3. Schools for Software Security: a Mapping Study

This section is a mapping study built upon the analysis of textbooks for software security. This mapping study summarizes the treatment deserved by cryptography in secure software development. The understanding provided by this study gave us the necessary perspective to disassemble traditional methods for developing secure software, identify gaps in current practice, propose areas for further investigation, and suggest domain-specific realizations aimed at cryptographic software.

We found that software industry supports distinct viewpoints for software security. These viewpoints differ in which of the available tools, techniques, or practices acts as the *remarkable feature* for the viewpoint. We name these viewpoints **software security schools**, because they are communities sharing similar ideas. Practitioners have to distinguish among instances of these schools and properly apply methods, tools, and techniques ac-

cording to the dominant philosophy governing specific endeavours for software development.

The three schools are the following. The **technology-driven** school is associated to primary software deliverables (e.g., code, designs, and tests). The **process-driven** school is characterized by threat modeling, defined processes, and vulnerability management. The **risk-driven** school is characterized by a risk mindset and security assessments for management supporting.

The section is organized as follows. Subsection 3.1 describes our strategy to this mapping study. Then, Subsection 3.2 describes the technology-driven school and Subsection 3.3 characterizes the process-driven school. Subsection 3.4 explains the risk-driven school, while Subsection 3.5 discusses similarities and differences among schools. Finally, Subsection 3.6 discusses how cryptographic software security is influenced by these schools.

3.1. Strategy to mapping study

We searched literature composed of books related to the broad subject of software security. We preferred textbooks concerned with the discipline of software security, because they give stable knowledge and show general practice already adopted by software industry.

Our search targeted two digital libraries: **Books24x7** [7] and **Safari Books** [8]. We choose these two libraries because they were promptly available to authors and were specialized in information technology, showing good collections for software development, software security, and cryptography, including titles from many publishers.

Searches were conducted in the first quarter of 2016. We adopted three keywords for individual searches: "software security", "secure coding", and "security engineering". When searching Books24x7, we found 1713 results for "software security", 1413 results for "security engineering", and 1078 for "secure coding". When searching Safari, we found 22.858 results for "software security", 17502 results for "security engineering", and 19824 for "secure coding". For both libraries and for each keyword, results were sorted by relevance and the list of search results was analyzed only up to page five (5), which means that only around the fifty (50) most-relevant results were selected for further analysis.

Next, in order to filter only those textbooks of interest, we applied a set of rules:

- Avoid books on specific technologies (e.g., patterns, web, python, cloud, Java, etc.).
- Avoid books exclusive for software development, information security, or cryptography.
- Avoid books on specific tasks or activities (e.g., coding, testing, or hacking).
- Avoid books that were collections of chapters from various authors, as well as handbooks.
- Avoid books older than 10 years, except when it was a classic reference mentioned by many.
- Avoid books older than 15 years anyway.

Then we were left with only those textbooks for software security. The relevant textbooks from Books24x7 digital library were the following. For keyword "software security": Talukder and Chaitanya (2008) [9], Ransome and Misra (2013) [10], Adam Shostack (2014) [11], Merkow and Raghavan (2010) [12], Viega, LeBlanc, and Howard [13], Douglas Ashbaugh (2008) [14], and Shon Harris (2012) [15]. For keyword "security engineering": Ross Anderson (2008) [16]. For keyword "secure coding": Richardson and Thies (2012) [17].

The relevant textbooks from Safari digital library were the following. For keyword "software security": Ransome and Misra (2013) [10], Viega, LeBlanc, and Howard [13], Viega and McGraw [18], Dowd, McDonald, and Schuh (2006) [19], and Gary McGraw (2006) [20]. For keyword "security engineering": Allen et al. (2008) [21], Ross Anderson (2008) [16]. For keyword "secure coding": Howard and LeBlanc [22], Howard and Lipner (2006) [23], Mano Paul (2011) [24], Chess and West (2007) [25].

This set of books was the base literature for the two mapping studies in this text. Also, the following books were found to be useful in general and complement the above lists: Daswani, Kern, and Kesavan (2007) [26], Hoglund and Mcgraw [27], Sommerville [2], the PMI Guide for Software [28], and the IEEE Guide for Software Engineering Body of Knowledge [29].

By reading the introductory chapters of all books, we could clearly identify three distinct discourses supporting distinct viewpoints for software security. We classified the selected textbooks, regarding those similarity of discourse, according to three schools. These viewpoints differ in which of the available tools, techniques, or practices is the remarkable feature according to the discourse. We

name these viewpoints as software security schools. For instance, three books are good representatives of each software security school: technology-driven [20], process-driven [23], and risk-driven [15]. The next subsections describe each school.

3.2. Technology-driven School

This school shows a technology-oriented view (a.k.a touch points [20]) and is associated to the primary deliverables of software development: source code, architecture design, tests, and executable binaries. The dominant belief of this school is that tools, techniques, and best practices have to be adopted in a pragmatic way, focusing on what developers actually produce during software development (e.g., source code and executable binaries).

The following textbooks share the same technology-oriented discourse and characterize this school: Gary McGraw (2006) [20]; Chess and West (2007) [25]; Daswani, Kern, and Kesavan (2007) [26]; Allen et al. (2008) [21]; Ross Anderson (2008) [16]; Talukder and Chaitanya (2008) [9]; and Richardson and Thies (2012) [17].

This school does not promote specific processes for secure software development. Instead, it enforces process enhancements, by hardening regular SDLCs with proper tools and techniques. The motivation for doing that is because a security-enhanced life cycle should compensate for security omissions in software requirements, by adding practices and checks to software development [20].

For this school, a secure development needs at least mechanisms for stopping the addition of known vulnerabilities and for monitoring changing security requirements [16], acting when needed.

3.2.1. Tools and Techniques

Automated support for secure coding and testing has been the common way to obtain a security-enhanced SDLC [20]. This school recognizes the importance of security tests, but prefers static analysis as the primary verification technique, because it can be applied early in development, as soon as developers start writing source code.

Static analysis is an approach to verification that examines the source code (or other representation) of a software system, looking for errors and anomalies [2]. It allows all parts of a program to be checked, not just those parts that are exercised by system tests. Static analysis tools were used to search source code against lists of known vulnerabilities, as well as quickly adapt to new issues [25]. However, static analysis tools frequently produce false positives (false alarms) and false negatives (omissions).

An advantage of security tools in general and of static analysis tools in particular is that they are more malleable than languages or frameworks, so the list of vulnerabilities identified by these tools can change faster than languages or frameworks can adapt [25].

When time and resource constraints prevent secure development practices from being applied to an entire software system, a business-driven risk assessment can determine which components receive highest priority [20]. However, security practitioners can be uncomfortable when forced to postpone implementation of security controls [16]. Risk analysis is particularly beneficial when applied to architecture and design [20].

Challenges in this school are to keep developers updated with attacks, to train them in useful skills, and to support them with appropriate tools [16]. Controversially, induce developers to think like hackers (or attackers) is not particularly favored by this school, because discovery of new attacks is supposed to be left to security researchers [16].

3.2.2. Technology-driven School into the SDLC

For requirements, techniques such as security checklists, threat modeling, and misuse cases have been used [9]. However, there is a need for methods to put security requirements into functional requirements [17]. Also, abuse/misuse cases showed little adoption [21].

For design, the Unified Modeling Language (UML) has been used as a modeling tool for secure design. However, UML is not a standard for architecture and allows only indirect representation of secure architectures [17]. Also, the adoption of attack patterns is not an agreement and security patterns were barely mentioned by textbooks.

For construction, authors agree that guidelines and coding standards raise awareness and teach secure coding. Besides general-purpose static analysis tools [25], other tools for secure coding can only be applied in specific domains [17].

Security testing should construct functional test cases to demonstrate the adherence to functional security requirements (positive requirements), develop risk-based test cases to exercise common mistakes and suspected weaknesses (negative requirements), and perform penetration testing [21].

For some authors [20, 25], code and design issues have to be treated first, even bypassing abuse cases, security requirements, and risk analysis. However, current practice emphasizes penetration testing and pushes software security to operational aspects, usually limited to simulated attacks [20]. This behavior may result from misunderstanding the idea that, before software is released, it should undergo a final security review with plans to incident response.

3.3. Process-driven School

This school follows Howard and Lipner [23] prescription for process-driven software security by emphasizing threat modeling, well-defined development processes, and vulnerability management. This school sees risk analysis as a result of threat modeling, not its remarkable feature.

The following textbooks share the same process-driven discourse and characterize this school: Howard and Lipner (2006) [23]; Merkow and Raghavan (2010) [12]; Mano Paul (2011) [24]; Ransome and Misra (2013) [10]; and Adam Shostack (2014) [11].

According to this school [12], the reliable way to ensure software is constructed securely is by integrating a security mindset and process view throughout the entire SDLC. Only this way, authors agree, flaws in processes can be avoided. However, the resulting engineering effort related to security may lead to an increment of around 20% of total development costs [23]. While costs for not implementing security may significantly exceed the costs for implementing it.

Still regarding costs, when other aspects of dependable systems have to be considered during development, the cost for verification and validation of such critical systems are usually much higher than for other classes of systems. For instance [2], more than half of a critical system's development costs are spent on verification and validation. Dependable systems encompass not only security, but also, safety, reliability, and availability [2].

For this school, SDLC processes have not only to correct bugs, but also to reduce the chances for inserting them [23]. Security is not a natural outcome of SDLC [10]. Thus, software security requires a focused effort to be effective: First, it needs to reduce the number of vulnerabilities and then reduces the severity of remaining vulnerabilities [23].

This school sees software security as the continuous management of risks and vulnerabilities into the SDLC [10]. For this school, handling security without specific processes can lead to frequent patching and divert developers' effort away from coding new features, directing them to respond to vulnerabilities in a reactive way [23].

3.3.1. Threat Modeling in Process-driven School

This school emphasizes threat modeling and understands risk assessment as a result of it, dividing risk management in two stages. First, a product risk assessment clarifies the effort required to fulfill requirements, helping determine how to spend resources to develop secure software. Then, during development, a system risk assessment determines the exposure to attack.

Because system understanding for threat modeling is challenging, time-consuming, subjective, and context sensitive, threat modeling should focus on issues other techniques cannot find, such as security flaws in architecture and business logic [11].

Threat modeling serves to understand the potential security threats to the system, determine risk, and establish appropriate mitigation controls [23]. Developers favor software-centric threat modeling against asset-centric risk assessment [11]. Roughly speaking, software-centric threat modeling starts with the creation of software models and ends with security issues being picked up, solved, and managed by normal development processes [11]. Two well-known methods for threat modelling can be found at the reference literature for this school [23, 30].

3.3.2. Process-driven School into the SDLC

This school starts to think security from requirements, trying to setup a security mindset with awareness and training activities [10].

Security requirements direct software architecture, complement threat modeling, and include resources like organizational policies, external regulations, and compliance obligations. Stakeholders should be engaged in requirements elicitation and contribute to a solid understanding, shared among project staff, about business decisions and risk implications [10]. Finally, cost analysis is also done as part of requirements analysis. Then, security requirements are documented and prioritized [10].

For secure design, authors complain that much attention has been given to secure coding and much less to secure design [23]. Surprisingly, authors also reject security patterns, because they have not been proved to be effective [11]. In spite of that, SSDLC

mandates that developers spend time in the design phase thinking about security of features and implementing secure designs.

Processes for secure design include design reviews for reduction of attack surface, threat modeling, identification and prioritization of security controls, and security architecture [24]. Also, design guidelines and best practices have been adopted [24].

This school values code analysis and peer reviews to mitigate or minimize code-level vulnerabilities. Also, secure coding is not only the correction of bugs or conformance to coding standards, but also includes version control and change management, which are both necessary to track changes and rollback to stable versions when needed [24].

In this school, security tests verify whether functions designed to meet non-functional requirements operate as expected, and validate whether implementation of these functions is not flawed [12]. Security testing should be performed by experts, never by users or developers, because, in general, developers are not good in uncovering flaws in their own code that are not related to operation [12]. Software is subjected to intense testing and, as resistance to attack increases, also increases the justified confidence that it is secure enough [12].

Security testing is iterative and results in lists of issues, ranked by risk and prioritized by stakeholders. Developers fix these issues and send the fixed code back to regression testing [12]. Before software is ready for deployment, it needs to be formally accepted, which usually requires a final assessment [24]. This last risk assessment, before release, empowers decision makers with justifiable confidence to accept software release [23].

3.4. Risk-driven School

This school is headed by both information security experts and security companies, whose businesses are software security. It enforces a risk management perspective to secure software development. However, it is conscious about the disadvantages of quantitative risk analysis for software.

The following textbooks share this risk-oriented discourse and characterize this school: Hoglund and McGraw (2004) [27]; Dowd, McDonald, and Schuh (2006) [19]; Doug Ashbaugh (2008) [14]; Ross Anderson (2008) [16]; and Shon Harris (2012) [15].

This school promotes a risk-oriented approach in order to provide strong evidence to regulators and auditors of due diligence in security management and governance [19]. The rationale is that real risk is hard to measure, because every environment has unknown vulnerabilities and persistent threats, but prioritizing risks according to which ones must be handle first is a realistic goal achievable in practice [15].

For this school, security is a business issue, because businesses sometimes need to operate risky to make money and have to handle security issues only if potential risks threaten their profit [15]. This school values skills to identify threats, assess their chance of occurrence, measure the resulting damage, and take steps to reduce risk to acceptable levels [15]. A frequent complain found in textbooks for this school is that businesses preferred security tools, detection/prevention of malicious code, and penetration testing, not fostering actual risk management [15].

3.4.1. Risk Management in Risk-driven School

Authors agree that it is barely possible to perform huge amounts of risk calculations, because accurate data about costs and probability of attacks is seldom known, as well as the time between vulnerability discovery and its exploitation is decreasing [14]. Thus, complex risk analysis to decide whether small mitigation controls to common code vulnerabilities must be put in place should be avoided [14]. Because, in the long run, simple controls will be put in place anyway.

Therefore, qualitative risk analysis is often utilized in software development efforts [14], not in the tiny scope of daily decisions, but for business-driven planning of activities. The goal of qualitative risk management in SSDLC is to reduce both the number of software vulnerabilities and the possibility of system compromise [15].

Risk management balances the protection requirements (for the organization and its assets) against the costs of controls to mitigate risks [15]. When software's functionality is limited by security constraints, market share and the potential profitability of that software could be reduced. Therefore, a balance always exists between functionality and security, and in businesses, functionality is usually preferred [15].

3.4.2. Risk-driven School into the SDLC

Frequently, software development is considered a temporary endeavor managed as a project. In this school, project management and risk management occur together [15]. For instance, at project's

initiation, preliminary risk assessment offers initial descriptions for security requirements [15]. At development, risk assessment identifies vulnerabilities and threats to the software and the potential risk [15]. At deployment, it is necessary to ensure the effectiveness of security controls and obtain formal authorization to deploy software to production [15]. In operation, vulnerability assessments, penetration tests, and remediation procedures often result in changes. Thus, configuration management and change control help to ensure that changes are tested and approved before deployment [15].

In this school, the integration of risk management and development processes encompasses not only security requirements to direct the implementation of safeguards and the adoption of proper development methods, but also the verification of mitigation effectiveness and the provision of secure and reliable distribution methods [15].

Into SDLC, risk management occurs as follows. In requirements, assets and threats are identified and an acceptance criteria for risk is defined to ensure that mitigation efforts are prioritized according to business needs [14].

In design phase, most threat modeling occurs to link software design to risk analysis [14]. Also in design, attack surface analysis is combined with threat modeling, in order to identify and reduce the amount of functionality accessible to untrusted users [15].

Coding activities result in the insertion of vulnerabilities, which are related to improper or faulty programming practices [15]. These vulnerabilities can be uncovered by mitigation strategies such as code reviews, pair programming, unit testing, and static analysis [14]. Also, mitigation strategies started in design (such as enforcement of coding standards) take effect during coding [14].

Simulated attacks and penetration tests identify missed vulnerabilities [15]. After tests are completed, a final risk assessment is conducted in order to make available to decision makers all information needed to securely authorize the deployment of a software release [14].

3.5. Discussion and Comparison

From an operational point of view, there is little difference among the three schools for software security, because they all share the same methods, tools, and techniques, which should be applied through the SDLC. Possibly, that is the reason why developers usually cannot see the differences. First, this subsection shows the similarities among schools and then discusses their differences. Table 1 compares the three schools.

3.5.1. Similarities among Schools

The three schools share the objective of systematically build secure software in an affordable way. Besides this intent, several common characteristics constitute the intersection between software engineering and software security.

Schools adopt two stages for risk management. First, a high-level, business-oriented risk analysis which helps in project planning and management. Second, a low-level, fine-grained risk analysis, sometimes called threat modeling or architectural security analysis, is software-centered and helps in both software understanding and security modeling.

Security requirements related to business logic are subjective and require specialized, time-consuming techniques, such as misuse cases, threat modeling, and business risk analysis. Frequently, these are neglected in favor of security checklists containing predefined controls (which are features, not requirements). Misuse cases are barely used.

Secure design is the most challenging phase for any school. Threat modeling, design reviews, architectural analysis, and attack surface reduction are subjective techniques intended to support the work of experts, who find design flaws by themselves. Best practices and principles for secure design have been adopted as high-level guidance, but requiring specialized interpretation to be put in practice. Security design patterns are not largely adopted.

Secure programming has been limited to coding standards and styles, which are enforced by language features, security frameworks, and static analysis tools. Schools agree that static analysis tools are the most cost-effective way to find well-known, code-level vulnerabilities. However, tools are incomplete, not overlapping, and buggy. Thus, various tools should be combined in order to obtain diversity and redundancy, promoting fault tolerance. Manual code inspection is usually adopted only for small portions of code.

All schools complain that software industry favors operational security of software, with a bias to penetration testing and final security assessments (before release), and does not favors software security in earlier phases of development. The incompleteness of tools is more evident for security testing than for other phases. In this matter, all

Table 1: Comparison of three schools for software security.

	Technology School	**Process School**	**Risk School**
Remarkable feature	Associated to main deliverables: source code, architecture, tests, and executables	Threat modeling, defined develop. processes, and vulnerability management	Security is business issue. Risk analysis evidences due diligence
SSDLC enforcement	Enforces process upgrade by hardening SDLC with security tools and techniques	Ensure software security by integrating security mindset and processes into SDLC	Joint project & risk management. Prior assessment of risk gives initial requirements
Requirements security	Security checklists, threat modeling, and misuse cases	Security requirements, awareness, and training complement threat modeling	identification of assets & threats. Scale for risk helps prioritize mitigation controls
Design security	Secure design, attack patterns, security patterns, and architectural analysis for security	Reviews reduce attack surface. Threat modeling, prioritized controls, and secure architecture	Threat model binds design & risk. Threat modeling and architecture assessment reduce attack surface
Construction security	Static analysis tools, code reviews, security guidelines, and standards for secure coding	Code reviews mitigate vulnerabilities, version control tracks changes and enables rollback	Common practice mitigates risk: reviews & inspections, unit tests, and static analysis
Testing security	Test cases for sec. requirements, risk-based tests for common vulns, and pen-testing	Security tests verify non-functional requirements, pen-tests for common vulns	Simulated attacks and penetration tests based on risk assessment and prioritization
Deployment & operations	Final security review before release and planning for incident response	Final risk assessment, before release, gives confidence to accept software	Final risk assessment needed to accept risk and authorize deploy or release

schools favor homemade testing tools and manual testing skills, but not vulnerability research.

3.5.2. Differences among Schools

The differences among schools are conceptual viewpoints and divide experts in comrades (fellows sharing the same ideas) or dissenters (who believe in something else).

The technology-driven school evolved to support developers in rapid cycles of development, where processes are not rigid and risk analysis is fast enough to not slow down the team, because risk is reduced progressively over time by the evolution of understanding. This school is good for highly adaptive life cycle [28], characterized by progressive specification of requirements, as well as short and iterative cycles of development. In organizations, this school is generally adopted bottom-up and conducted by technical staff (with support from executives). A possible scenario for this school emerges when a lower manager enhances the software security capabilities of his team by integrating into the SDLC a well-crafted set of tools and techniques.

The process-driven school evolved in places with a bias to project management. This school is good for highly predictive life cycles [28], which emphasize specification of requirements and detailed planning in initial phases of development, because detailed plans based on known requirements reduce risks. In this school, long-term development cycles and long-lived software products provide an adequate environment for information gathering about vulnerability reductions over time, as well as recordings of historical information about vulnerability severity and security compromises. A possible scenario for this school occurs, for instance, when a middle manager uses status meetings, metrics, and process milestones to follow progress of software development projects.

The risk-driven school evolved in organizations where governance and compliance have great influence on software development, as well as risk management is imposed to software development (top-down). This school aims at providing executives with enough evidence of due diligence concerning

software security in particular, and operations security and business risk, in general. A possible scenario for this school happens when an upper manager periodically receives visits of external auditors and has to be prepared with enough evidence of due diligence to comply with regulations.

Schools may not occur in isolation and blended approaches are possible. Furthermore, blended approaches do not discard the current school, but enriches it with new ideas from other schools. For instance, organizations accustomed to highly predictive life cycles, when pushed by time to market, have to adapt to highly adaptive life cycles. Also, predictive life cycles are adopted by software factories having to fulfill compliance requirements and technology companies having to accomplish stock market obligations for business governance.

Finally, authors [2, 20] start to think about security as an emergent property of software systems. Software security is not made only of features to be added, and security features alone are not sufficient for building secure software [20]. Instead, security may be understood as an emergent property of software systems [31]. In order to handle the emergence of security properties, a systems-theoretic approach to software security has been suggested [32] as a paradigm shift in software industry from vulnerability management to security engineering and architecture.

3.6. The Case for Development of Secure Cryptographic Software

We have seen that the development of cryptographic software suffered great influence from communities, which ultimately determined how cryptography is approached by software security schools.

In general, software security does not directly address the issues of cryptographic security. Because, cryptography is considered a security feature effectively added to software during coding, with no specific support from development processes [16, 20]. For many experts, cryptography issues reside only below APIs, inside libraries, concerned only with the security of algorithm implementation. For instance, in a survey on advanced tools for coding and verification of cryptographic software [33], we found that only one quarter (around of 25%) of all surveyed tools can be applied above cripto APIs. The other 75% being applicable only below APIs, for algorithm implementation.

This bias towards algorithm implementation, instead of its correct use, contributed to perpetuate misunderstandings about the correct use of cryptography, resulting in the exploitation of simple vulnerabilities with catastrophic consequences (see [34, 35]), as well as leading to frequent misuse of cryptography [36–38], improper certificate validation [39, 40], and inappropriate error handling when orchestrating cryptographic services [41–43].

Many studies showed that vulnerabilities in crypto software are mainly caused by software defects and poorly managed parameters [44–47]. Recent studies [36–40, 48, 49] showed the occurrence of known crypto misuses in modern software platforms. Also, current tools are unable to cope with security issues in programming crypto software [50]. Furthermore, most advice regarding cryptography found in textbooks for software security is obsolete by the standards of today (as we explain in next section).

Ross Anderson [16] suggests that there is a social divide between the two communities of cryptology and computer security. The computer security and cryptology communities have drifted apart over the years because security experts do not always understand cryptographic tools, and cryptologists do not always understand real-world problems [16]. Rogaway [51] claims that cryptologists should acquire a system-level view and attend to what surrounds their field. Also, Green and Smith [52] argue that modern security practice has created an adversarial relationship between security software designers and developers. For them [52], security experts must focus instead on creating developer-friendly approaches to strengthen systems security (in opposition to a developer-proof attitude [53]). Rogaway [51] suggests that one approach that might be useful is to take an API-centric approach. For him [51], API misunderstandings are a common security problem, and (semantic) gaps between cryptography theory and API design can produce serious problems for cryptographic software.

We perceive an echo of this social divide when experts in software security systematically avoid entering cryptography issues during development, minimizing them to high-level, theoretic knowledge and punctual advice from experts. For instance, the Open Web Application Security Project (OWASP) [30] recognizes a current focus on high-level guidelines for developers and architects who implement cryptographic solutions. Software security favored penetration testing against known

vulnerabilities of standard protocols (e.g., Heartbleed [34]), enforcement of simple coding guidelines against misuse of crypto libraries (e.g., dot not use DES or MD5), and compliance to general security policies (e.g., always use TLS with HTTP).

Finally, we argue that, regarding cryptographic software, the complex relationships between the components in a cryptosystem mean that it is more than simply the sum of its parts. A cryptosystem embedded in modern cryptographic software manifests what Sommerville [2] calls functional emergence, when the purpose of a system only emerges after its components are integrated. For instance, in general, the parts of a simple cryptosystem for symmetric encryption are key management, randomness sources, encryption algorithms, protocols for key agreement or distribution, configurations for key length and other parameters, as well as how all these parts are blended to business logic.

4. Cryptography Misuse in Software Security: a Mapping Study

In general, software developers are exposed only to good uses of cryptography, in simple examples related to common use cases. However, cryptography is full of potentially dangerous design decisions and coding pitfalls that confuse the ordinary developer, usually a non-expert in cryptography, when developing actual cryptographic software.

On the other hand, as said before in previous section, it is not good practice in software security to teach developers how to attack systems or give them incentives to research new vulnerabilities. This way, cryptography misuse show up as an intermediary alternative that sits somewhere in between misuse cases of system behavior and attack patterns for cryptography.

Therefore, we propose cryptography misuse as a novel way to educate developers on how to avoid all those design flaws and coding pitfalls frequently related to cryptographic software, increasing awareness in the correct usage of cryptography, although, without requiring from developers an adversarial mindset.

The development of secure cryptographic software requires a classification of recurring misuses of cryptography in software, over which methods, tools, and techniques could be built upon. A thorough understanding of the characteristics of cryptography misuse, captured by a classification, is required to design effective tools for preventing, detecting, and mitigating these misuses, also providing a background to appropriately position research activities.

For instance, by determining what kinds of crypto misuse can be found more frequently in communities of developers, we are able to investigate how they appear in software, how they are covered by current security tools, and how they relate to each other.

The main contribution of this section is a classification for cryptography misuse from a software security point of view. The proposed classification emerged from a literature review (mapping study) performed in an iterative and incremental process. First, we reviewed selected literature for software security that also cover cryptography issues, and adopted their labels, grouping them in the main categories with few sub-items. Then, new categories were added and sub-items were refined from industry sources and recent studies. Validation and further refinement occurred by studying online communities for cryptography programming as well as by evaluating static code analysis tools. This mapping study is a refinement from the previous mapping performed in Section 3.

This section is organized as follows. First, Subsection 4.1 describes related works concerning other classification efforts for computer security, and how we selected the literature to support the mapping study and the design of the classification. Then, Subsection 4.2 gives a detailed description of our classification and Subsection 4.4 discusses limitations and design decisions of our approach.

4.1. Related Work and Supporting Literature

During the previous decades, a few lists and taxonomies of security problems have been developed for various purposes. Unfortunately, none of the existing works organized cryptography misuse by characteristics that were useful for our needs.

Landwehr et al. [54] proposed a taxonomy for security flaws in computer programs by collecting and organizing a number of actual security flaws in different operating systems and classifying each flaw according to its genesis, the time it was introduced into the system, or the section of code where each flaw was introduced. Aslam, Krsul, and Spafford [55] proposed a classification of security faults in the Unix operating system for helping in the unambiguous classification of security faults suitable for data organization and processing by tools.

Weber, Karger, and Paradkar [56] combined previous categories of security problems and reports on security incidents in order to create a single taxonomy for security flaws. They also correlated their taxonomy with current threats, suggesting that their taxonomy is suitable for tool developers. Tan et al.[57] studied bug characteristics in three open-source projects, finding that semantic bugs are the dominant root cause. They also suggested that more support is needed to help developers diagnose and fix security bugs, especially semantic security bugs.

Wan et al. [58] studied the bug characteristics in eight open source systems for blockchain, examining bug reports, categorizing bugs, and investigating the frequency distribution of bug types across projects and programming languages of this technology. This is an example of a domain-specific classification of bugs and vulnerabilities.

None of the above-mentioned classifications for security flaws focus on cryptography misuse from a software security point of view. Therefore, a domain-specific classification was still needed to support our work. However, we were not interested in advice for secure implementation of cryptographic algorithms, such as the ones found in specialized resources [59, 60]. Instead, we looked for programming techniques for building secure cryptographic software. On the other hand, the field of cryptography is diverse and a rough analysis of its literature landscape showed the existence of niches.

For instance, there are books on applied cryptography [61–63] and cryptographic engineering [60, 64], which are intended for scientists and engineers actually embedding cryptographic algorithms into hardware appliances or securely implementing cryptography in software libraries. Also, there are books on cryptographic APIs [65, 66] and programming [67, 68], which teach how to use cryptographic software packages or libraries in specific programming languages. In general, these books do not focus on cryptography misuse, but are mainly targeted at specific APIs and their coding tricks, giving little advice on proper use of cryptography.

Additionally, some books on software security do cover security issues related to cryptography misuse. Sometimes, they give *ad-hoc* lists of secure programming practices [13, 18, 22, 25, 26]; sometimes they also give advice for architectural decisions [11, 16, 23]. Also, there are good-standing, online resources trying to give practical advice on cryptography usage. For instance: the top-10 security risks [69] and security testing guide [30] from OWASP; the top-10 secure design flaws [70] from IEEE Cybersecurity Initiative (CYBSI); the practices for secure software development [71] from the Software Assurance Forum for Excellence in Code (Safecode); and SANS's top-25 most dangerous programming errors [72].

In summary, to identify cryptography misuse and related advice aimed at software security, in this mapping study, we searched mainly through three source classes:

1. Literature on software security that also covers cryptography issues (e.g., [11, 13, 18, 22, 23, 25, 26]).
2. Industry initiatives for software security (e.g., [70–73]).
3. Studies on cryptography misuse by software developers (e.g, [36–40, 53, 74–76]), including analysis of complex misuses that are starting to become known by developers (e.g., [42, 77–80]).

4.2. A Classification of Cryptography Misuse for Software Security

The classification of cryptography misuse proposed in this section captures how software developers actually misuse cryptography. Therefore, it is inclined to a software security viewpoint. The classification has nine main categories: Weak Cryptography (WC), Bad Randomness (BR) handling, Coding and Implementation Bugs (CIB), Program Design Flaws (PDF), Improper Certificate Validation (ICV), Public-Key Cryptography (PKC) issues, IV/Nonce Management (IVM) issues, Poor Key Management (PKM), and Cryptography Architecture Issues (CAI). These categories were synthesized from recommendations covered by literature and are supposed to capture the state of practice.

Additionally, we identified three qualitative groupings for the nine categories (in Table 2) that can be associated to software abstractions (e.g., code, design, system architecture) required to actually understand the issue.

In this text we adopt the ideas of McGraw [20] to distinguish among code-level issues, design flaws, and insecure architectures. The distinction is derived from three factors: *(i)* how much source code must be considered to understand the crypto misuse, *(ii)* how much detail regarding the execution environment must be known to understand the

crypto misuse, and *(iii)* whether a design description is best for determining whether or not a given crypto misuse is present.

For instance, design-level crypto misuse involves interactions among more than one location in code and configurations, and architecture-level crypto misuse carry this trend further, considering execution environments and software platforms. The groupings are as follows:

1. **Misuse Group One (MG1)** is related to low-complexity issues in coding and in APIs, and could be easily found by early detection techniques, simple code reviews, and skilled developers (supported by tools). It includes Weak Crypto (WC), Coding and Implementation Bugs (CIB), and Bad Randomness (BR) handling. No deep understanding of program design is required to mitigate crypto misuse in coding, because fixes are likely to be simple and related to single programs or simple code snippets.
2. **Misuse Group Two (MG2)** is related to medium-complexity flaws in program design affecting a few different programs and may be difficult to identify due to feature distribution across programs. It includes Improper Certificate Validation (ICV) issues, Program Design Flaws (PDF), and Public-Key Crypto (PKC) issues. Fixing these misuses may require program redesign and may affect a few programs. Avoiding them requires more knowledgeable developers and support from experts.
3. **Misuse Group Three (MG3)** is related to high-complexity flaws in system design and architecture, and requires understanding of system architecture to analyze underlying cryptosystems. It includes Poor Key Management (PKM), IV and Nonce Management (IVM) issues, and Crypto Architecture and Infrastructure (CAI) issues. Fixes in this group usually require new modules or redesign of modules, and may affect many code bases. These misuses require cryptography experts to perform code and design reviews, or architecture analysis.

The next subsections detail misuse categories and their sub-items. Table 2 details the descriptive subsets for cryptography misuse, mapping them to literature.

4.2.1. Weak Cryptography

This category consists mainly in the programmatic misuse of obsolete, broken, or misconfigured cryptography; in particular, encryption algorithms, hash functions, and message authentication codes (MACs). We found that most authors for software security associate the broad words of "cryptography" or "encryption" with symmetric encryption, putting public-key cryptography in other category. This classification tries to capture this perception in order to facilitate the communication with software developers.

For Viega and McGraw [18] and Daswani, Kern, and Kesavan [26], cryptography practitioners should not invent their own cryptographic algorithms or protocols. Instead, these authors recommend to stick with well-scrutinized protocols and to use well-scrutinized implementations of these protocols. Howard and LeBlanc [22] strongly recommend ordinary developers to not create proprietary encryption algorithms, because of the great chances of getting it wrong. For Howard and Lipner [23], software projects should use standard cryptographic libraries, as well as standard high-level protocols, rather than low-level cryptography.

Chess and West [25] recommend that a programmer should not invent nor implement cryptographic algorithms or (key exchange) protocols. For both Howard, LeBlanc and Viega [13] and Adam Shostack [11], instead of building proprietary protocols from low-level algorithms, high-level, well-tested protocols should be adopted.

Selection of insecure algorithms is other way to get weak cryptography. Algorithm selection is an example on how software security has evolved at a slower pace than applied cryptography. For instance, Howard and LeBlanc [22] enforce the use of RC4, and Howard and Lipner [23] recommend that block ciphers should always use CBC mode. These advice are obsolete by the standards of today, but have been adopted by other authors (e.g., Daswani, Kern, and Kesavan [26]) as well as software developers as relevant for software security.

Howard and Lipner [23] demand AES for new code, three-key 3DES for backward compatibility, older block ciphers for decrypting old data only, and banishment of RC4. For hashing, they [23] advice SHA-1 for backward compatibility and SHA-2 for dotNET and server-side code. Chess and West [25] recommend AES for encryption and RSA for secret keys exchange and digital signatures. Howard,

Table 2: A classification of cryptography misuse for software security.

	Category	Sub-type or sub-item	Literature source
MG1: Low complexity	Weak Cryptography (WC)	- Risky or broken encryption - Proprietary cryptography - Determin. symm. encryption - Risky or broken hash/MAC - Custom implementation	[38] [74] [36] [37] [75] [18] [22] [25] [13] [23] [71] [73]
	Coding and Implementation Bugs (CIB)	- Wrong configs for PBE - Common coding errors - Buggy IV generation - Null cryptography - Leak/Print of keys	[38] [74] [36] [75] [81] [13] [73]
	Bad Randomness (BR)	- Use of statistic PRNGs - Predict., low entropy seeds - Static, fixed seeds - Reused seeds	[38] [74] [36] [75] [18] [22] [25] [13] [23] [71] [73]
MG2: Medium complexity	Program Design Flaws (PDF)	- Insecure default behavior - Insecure key handling - Insecure use of streamciphers - Insecure combo encrypt/auth - Insecure combo encrypt/hash - Side-channel attacks	[38] [74] [37] [45] [75] [18] [22] [23] [11] [71] [73] [42]
	Improper Certificate Validation (ICV)	- Missing validation of certs - Broken SSL/TLS channel - Incomplete cert. validation - Improper validated host/user - Wildcards, self-signed certs	[38] [37] [39] [40] [75] [11] [73]
	Public-Key Cryptography (PKC) issues	- Deterministic encrypt. RSA - Insecure padding RSA enc. - Weak configs for RSA enc. - Insecure padding RSA sign. - Weak signatures for RSA - Weak signatures for ECDSA - Key agreement: DH/ECDH - ECC: insecure curves	[74] [36] [37] [45] [75] [77] [25] [23] [78] [79]
MG3: High complexity	IV and Nonce Management (IVM) issues	- CBC with non-random IV - CTR with static counter - Hard-coded or constant IV - Reused nonce in encryption	[74] [36] [37] [75] [53] [76]
	Poor Key Management (PKM)	- Short key, improper key size - Hard-coded or constant keys - Hard-coded PBE passwords - Reused keys in streamciphers - Use of expired keys - Key distribution issues	[38] [74] [36] [75] [18] [22] [25] [13] [23] [11] [70] [73]
	Crypto Architecture Issues (CAI)	- Crypto agility issues - API misunderstanding - Multiple access points - Randomness source issues - PKI and CA issues	[38] [74] [36] [37] [45] [40] [75] [18] [13] [23] [11] [71] [73]

LeBlanc, and Viega [13] recommend AES and SHA-2, and warn against insecure cryptography (e.g., MD4, MD5, SHA-1, DES, RC4, and ECB mode). Daswani, Kern, and Kesavan [26] recommend AES, but also DES, 3DES, ECB mode. Daswani, Kern, and Kesavan [26] suggests MD5, SHA-1, and SHA-2 (the only good choice) for secure hashes.

Concerning weak cryptography, Safecode [71] strongly advises developers to prefer standardized security technologies that have undergone public review, rather than using low-level cryptographic algorithms or developing custom cryptographic protocols. For insecure selection of cryptographic algorithms (a subset of weak cryptography), Safecode [71] and OWASP [73] warn that the following algorithms and cryptographic technologies should be treated as insecure: MD4, MD5, SHA1, DES (and its variants), RC4 (and other stream ciphers), block ciphers in ECB mode, and any cryptographic algorithm that has not been subject to open academic peer review.

Still regarding weak cryptography, according to Egele et al. [36] and Shuai et al. [37], the most common misuse is symmetric deterministic encryption, when a block cipher (e.g., AES or 3DES) uses Electronic Code Book (ECB) mode. There are cryptographic libraries in which ECB mode is the default option, automatically selected when the operation mode is not explicitly specified [76].

4.2.2. Bad Randomness

This category consists in the programmatic misuse of Pseudo-Random Number Generators (PRNGs), mainly formed by cryptographically insecure PRNGs, which should not be used for cryptography matters, or insecure seeding of secure PRNGs.

Viega and McGraw [18] recommend the use of many entropy sources, preferring hardware, to promote fault tolerance. For them [18], enough randomness can be obtained from small amounts of good entropy stretched through a PRNG. Howard and LeBlanc [22] warn that statistical PRNGs produce predictable sequences. Chess and West [25] explain that common errors occur when statistical PRNGs are used instead of cryptographic PRNGs, or a secure PRNG is seeded with insufficient entropy (less than 64 bits [13]).

Daswani, Kern, and Kesavan [26] agree with all these advice. For Howard, LeBlanc and Viega [13], programmers should not build PRNGs, because most operating systems today come with good enough PRNGs. For instance, Oracle provides a comprehensive list of PRNGs available for Java programmers in many operating system [82].

Concerning bad randomness handling, Safecode [71] recommends developers to use high quality PRNGs when creating cryptographic secrets (e.g., encryption keys), and avoid algorithmic (statistic) random number generators in cryptographic code. A misuse related to bad randomness is hard-coded or constant seeds for PRNGs [36].

4.2.3. Coding and Implementation Bugs (CIB)

This category consists of several insecure coding practices specific to cryptographic software, such as misconfigured Password-Based Encryption (PBE), leakage of keys (e.g., printing), and incorrect handling of binary data (e.g., saving cipher text as strings). The most mentioned coding bug related to cryptography misuse is misconfigured PBE. For instance, Shuai et al. [81] discovered that password protection in Android is greatly affected by cryptography misuse, such as PBE. When using passwords to derive cryptographic keys, Howard and LeBlanc [22] simply advice to make sure that passwords are long enough and highly random, as well as to consider a balance between randomness and ease of recall. For Howard and Lipner [23], a Key Derivation Function (KDF) should always be used to derive keys from passwords. Those authors only provide general advice and no practical misuse.

Howard, LeBlanc and Viega [13] advise that a good KDF, such as PBKDF2 (RFC 2898), should be used when creating a password verifier. Also, they recommend that the number of iterations (when using iterated hash) should not be less than 1,000, preferring greater values (100,000 iterations) in modern systems. Also, salts should be randomly chosen and preferably not less than 16 bytes.

For Safecode [71], vulnerabilities come from key exposure via insecure software. For instance, when private or secret keys are printed to consoles or save to log files. Safecode [71] explains that, while at rest, keys should always be managed within a secure database, a secure file system, or hardware storage.

4.2.4. Program Design Flaws (PDF)

Program design flaws include crypto misuses related to incorrect use of stream ciphers, insecure combinations of encryption and hashes or MACs [53], as well as side channels due to padding oracles [42] and timing channels in verifying hashes for passwords or Message Authentication Codes

(MACs) [48]. Adam Shostack [11] asserts that verification of hashes and MACs should be done in constant time.

Interestingly enough, recommendations for combining hashes and MACs were made before the advent of atomic authenticated encryption (AE). Thus, many authors do not detail ways to mix MAC and encryption, which, by the way, should be avoided in favor of AE functions [53].

Viega and McGraw [18] advise to use a MAC for message integrity anytime encryption is used and carefully handle failed verification, exceptions, and errors. Adam Shostack [11] asserts that all cryptographic systems should verify authenticity of encrypted messages before decryption. Howard, LeBlanc and Viega [13] explain that concatenating user-generated data before hashing is insecure, because it allows for unauthorized handling of hashes by adversaries. In case of insecure combination of MACs and encryption, only OWASP [73] advises to calculate a MAC over encrypted data (Encrypt-than-MAC), the least dangerous way, and to adopt AE functions, but fails to show how to do it.

Insecure handling of keys is a kind of program design flaw. Howard and LeBlanc [22] advise to keep keys close to the point where they encrypt and decrypt data, because the more code has access to secret data, the greater the chance secrets are compromised. Also, secret data or keys passing throughout applications are more likely to be compromised than secret data kept and used only locally. For Safecode [71], this misuse is mitigated when access to keys is granted explicitly via access control mechanisms. Also, after retrieving keys from secure storage, applications should not persistently store them elsewhere. Then, when keys are no longer needed, they must be securely erased.

Insecure use of stream ciphers is other design flaw. For Viega and McGraw [18], poor understanding of cryptographic primitives is the cause of key reuse in stream ciphers (or similar block cipher modes). Howard, LeBlanc and Viega [13] recognize that developers often use stream ciphers incorrectly. Howard and Lipner [23] warn that deployed stream ciphers should always undergo security reviews.

4.3. Improper Certificate Validation (ICV)

This crypto misuse category consists of incorrect validation of digital certificates by application software. Sometimes the proper validation is made impossible, because digital certificates were generated in insecure ways. For example, in the case of wild-card or self-signed certificates.

In general, software security books do not talk about the dangers and pitfalls of improper certificate validation. On the other hand, several recent studies [37–40, 83] showed that libraries for handling SSL/TLS connections has several issues when validating digital certificates. Georgiev et al. [39] and Fahl et al. [40, 83] showed that these libraries allow programmers to ignore parts of certificate validation in favor of usability or performance, but adding vulnerabilities to the application. Also, failures in signature verification or domain-name validation facilitate man-in-the-middle attacks [39, 40].

For OWASP [73], it is important to ensure that certificates are properly validated against the host-names or users, as well as to avoid using wild-card certificates.

4.3.1. Public-Key Cryptography (PKC) issues

This crypto misuse category contains all those design flaws related to encryption and signing with public-key cryptography. We found that software security books treat public-key cryptography separated from symmetric cryptography.

Concerning public-key cryptography, Howard and Lipner [23] recommend the algorithms RSA, ECDSA, and DH for digital signatures, as well as NIST curves P-256, P-384, and P521 for Elliptic Curve Cryptography (ECC). Daswani, Kern, and Kesavan [26] suggests ECC and RSA for asymmetric encryption.

Asymmetric deterministic encryption is a misuse related to non-randomized RSA [45]. Also, PKC issues include weak or misplaced parameters for RSA [77], misconfiguration of key agreement protocols (e.g., DH and ECDH) [80], and ECC [78, 79] misuses, concerning both the selection of insecure curves, and the reuse of nonces for ECDSA.

4.3.2. IV and Nonce Management (IVM) issues

This crypto misuse category is formed by design flaws and insecure architectures regarding management of Initialization Vectors (IVs) and nonces (numbers used only once) as complementary aspects for cryptosystem. We found that IV and nonce management is other cryptography misuse that software security books do not talk about. However, this misuse have been discovered by recent studies.

For instance, hard-coded IV has been considered a frequent misuse [36]. In several operation modes

of block ciphers, IVs must be unique and unpredictable, the Counter (CTR) mode requires unique IVs (without repetition), these options are misused by developers [53]. Still, other misuses come from exchanging operation modes without considering IV requirements [48, 53]. For instance, Java Cryptographic Architecture (JCA) [82] allow operation modes to be easily changed.

4.3.3. Poor Key Management (PKM)

This crypto misuse category consists in the insecure management of cryptographic keys and related (sensitive) material through out a key's life cycle: creation, assignment, activation, (operational) use, expiration, revocation, and destruction. This category is related to insecure architectures because, frequently, in order to fix key management issues, whole modules have to be build. For instance, a hard-coded key found in source code means that a module for key management is missing.

For Howard and LeBlanc [22], all too often, good systems are let down by poor key management, because securely storing and using keys is hard. They [22] strongly warn against hard-coding secret keys in (source) code or executables. Most software security authors are not concerned with proper key lengths, with the exception of Howard and Lipner [23], despite their outdated advice. Chess and West [25] and Daswani, Kern, and Kesavan [26] barely mention it by suggesting RSA with at least 1024-bit keys.

Adam Shostack [11] explains that key management must ensure that each party gets the right keys, and that there's some mapping between keys and accounts or roles within a system. He argues [11] that developers sometimes have no option but mistakenly manage keys either locally or manually, due, for instance, to infrastructure issues that lead to poor validation of digital certificates. He prefers [11] asymmetric cryptosystems for key distribution. However, keys still need to be authenticated.

Insecure use of stream ciphers may result from reuse of keys. Viega and McGraw [18] explain that stream ciphers and block ciphers in stream modes (e.g., OFB, CFB, and CTR) do not preserve data integrity because cipher text is malleable. Howard and LeBlanc [22] recommend randomized modes of operation combined with keyed-hashes, as well as warned against key reuse in stream ciphers. All these authors fail to advise against reuse of keys or IVs. Only Daswani, Kern, and Kesavan [26] mention the general issue of hard-coded keys, but not for the specific context of stream ciphers.

IEEE CYBSI [70] warns that, ultimately, the security of cryptosystems still hinges on the protection of keys. CYBSI [70] warns that the following mistakes are common: hard-coded keys in software, failed revocation of keys, short or predictable keys, and weak distribution mechanisms. Safecode [71] considers that keys have very high security requirements, which should be assured throughout the life cycle of the key.

4.3.4. Cryptographic Architecture Issues (CAI)

This crypto misuse category consists in all those wrong architectural choices regarding management of cryptographic infrastructures (cryptographic libraries, PRNGs, secure storages, etc.) and how they are made available to ordinary developers.

Viega and McGraw [18] prefer well-known cryptographic libraries and stable technologies instead of unknown or experimetal ones. For them, for instance, SSL is an important technology asset for online security, despite its documented implementation problems, and should be preferred to achieve secure communication, instead of manually crafting secure channels from low-level cryptographic functions.

Howard and Lipner [23] use the term *cryptographic agility*, when software provides rapid and simple ways to upgrade cryptographic algorithms over time. Additionally, if the software uses multiple algorithms to maintain backward compatibility, it must not silently default to obsolete or deprecated ones. For Howard, LeBlanc and Viega [13], cryptography agility is achieved when a software accommodates updates of encryption algorithms that developers had not anticipated, and still works correctly.

On the other hand, Adam Shostack [11] argues against excessive flexibility. For him [11], it is common for developers to wrongly include flexibility and negotiation of parameters when designing a protocol (e.g., when blending cryptography into application's functionality). However, such negotiation augments attack surface and exposes communication to man-in-the-middle attacks [11].

Still, for cryptographic architecture issues, Safecode [71] argues that applications should reuse cryptographic functions as a service, avoiding common mistakes due to implementation of proprietary cryptography. For CYBSI [70], developers should

not assume that just using strong libraries will be enough. CYBSI [70] warns about the misuse of libraries and algorithms and explains that understanding the nuances of algorithms and library usage is a core skill for applied cryptographers.

CYBSI [70] also warns about the failure to centralize cryptography (keeping multiple implementation of the same algorithm) and to allow for algorithm adaptation and evolution, because proper interaction among different implementations is always a concern. For secure cryptographic storage, OWASP [73] advises that an architectural decision must be made to determine the appropriate method to protect data at rest. Furthermore, a standard or policy must be kept to ensure that developers know about approved choices for protocols, algorithms, as well as cryptoperiods, and key management infrastructures.

4.4. Design decisions and limitations

This subsection discusses design decisions that drove the structure of our classification and points limitations to the work.

A taxonomy is not simply a neutral structure for categorizing items, because it implicitly embodies a theory of a field from which those items are drawn and distinguished [54]. For us, a classification embeds a viewpoint. By creating a classification of cryptography misuse for software security, we also create a theory of such misuses that organize a field of study and help to answer software security questions from a collection of misuse instances.

Because crypto misuses were not randomly selected from a valid statistical sample of software flaws and vulnerabilities, we make no strong claims concerning the likely distribution of actual crypto misuses in software. On the other hand, we do have evidence [48, 49] for the frequency of cryptography misuse in online communities for cryptographic software programming and how likely crypto misuses are related to use cases and coding tasks.

A populated classification contributes to the understanding of what actually causes security breaches [54]. Therefore, in order to satisfy the need for actual instances of crypto misuses, we performed empirical studies as well as experiments that validated the classification. We do not suggest that we have assembled a representative random sample of all known cryptography misuse, but we did our best to include a wide variety of them, which are supported by a code base filled with realistic instances [50].

Taxonomies like ours are more relevant to designers of tools as well as for code inspection [56]. Because we took the viewpoint of the developer, we suggest our classification can help design better cryptographic software.

The classification is not simply a list of vulnerabilities. In fact, it has to be designed with a goal in mind. For example, improper use of cryptography or under-specified crypto APIs are ease to detect in source code by code analysis techniques. On the other hand, the detection of design flaws is more difficult, because programmers intent may not be reflected in code in convenient ways. We decided that our classification would include only those design flaws detectable through code inspection (either manual or automated) than through other methods. That is the reason why we needed to identify actual instances of these misuses and group them according to the level of system abstraction required to actually see the misuse: source code, program design, or system architecture. The very existence and importance of these groupings were validated by empirical studies and experimental analysis described in the next sections.

5. Empirical evidence for cryptography misuse

This section provides empirical evidence to support the classification of cryptography misuse for software security. We overview two complementary empirical studies [48, 49] about cryptography misuse in online communities for cryptography programming. First [48], we analyze the occurrence of cryptography misuse in two online communities by the contributions of their users (software developers). Second [49], we investigate whether active developers participating in these two online communities are getting better in using cryptography with time.

Developers are frequent users of online communities for programming. In general, the agility in problem solving provided by many question-and-answer communities brings benefits to ordinary programmers lacking knowledge in specific topics, such as secure coding or cryptography.

In these studies, we selected two programming communities supported by experts in applied cryptography: Oracle Java Cryptography (OJC) [84], a forum aimed at programming with Java Cryptographic Architecture (JCA) [82], and Google Android Developers (GAD) [85], a popular forum

for Android programming. We picked these two communities because they share the same Java-based API for the Java Cryptographic Architecture (JCA) [82], thus limiting the knowledge required by a code reviewer to four aspects: Java programming, JCA, Android security, and applied cryptography. Also, JCA offers a stable and generic API, which has been used for a long time by a large group of developers for both server-side applications and mobile devices. Furthermore, JCA was adopted by the Android platform as its main API for cryptographic services. These two communities together reach a large number of ordinary developers, most of them are supposed to be non experts in cryptography.

We found that, several types of cryptography misuse can be found frequently in online posts (90% for Java and 71% for Android), but were masked by technology-specific issues. We also found that different types of cryptography misuse were statistically related, frequently appearing in double or triple associations. Then, we found that cryptography misuse was not only common in these coding communities, but also recurrent in developer's discussions, suggesting that developers learned how to use crypto APIs without actually learning the tricky details of applied cryptography.

Subsection 5.1 explains how cryptography misuse appears in online communities, while Subsection 5.2 discusses how developers use crypto APIs without learn cryptography. Then, Subsection 5.3 summarizes our findings.

5.1. Cryptography misuse in online communities

We observed that OJC community suffered the most influence from weak cryptography (WC, 26%), architectural issues (CAI, 20%), coding bugs (CIB, 17%), public-key issues (PKC, 16%), and poor key management (PKM, 11%). Also, in OJC, architectural issues were strongly related to platform-specific issues (CAI&PSI) due to complexity of Java's cryptographic architecture. Furthermore, weak cryptography followed by coding bugs (WC&CIB) or public key issues (WC&PKC) were other easily recognized patterns. Mistakes in key management were also related to weak cryptography (WC&PKM), coding bugs (CIB&PKM), or public-key issues (PKM&PKC). Interestingly, OJC showed occurrences of two triple patterns: a rule formed by weak cryptography, coding bugs, and design flaws (WC&CIB&PDF) and a rule formed by weak cryptography, coding bugs, and key management (WC&CIB&PKM).

The GAD community suffered most from weak cryptography (WC, 21%), coding bugs (CIB, 17%), and public-key issues (PKC, 10%). This behavior was probably due to API misunderstanding and lack of knowledge in cryptography programming. GAD presented a more association rules than OJC. For instance, the most prevalent rules included weak cryptography associated to coding bugs (WC&CIB) or platform specific issues (WC&PSI). Also, Android specific issues were related to coding bugs (CIB&PSI), public-key issues (PKC&PSI) and design flaws (PDF&PSI). Coding bugs were also related to issues in IV management, (IVM&CIB).

Also, GAD showed four triple patterns, all of them affected by Android specific issues: Weak cryptography with Android specific issues and coding bugs (WC&PSI&CIB), weak cryptography with platform issues in public-key crypto (WC&PSI&PKC), coding bugs with platform issues in IV management (CIB&PSI&IVM), and design flaws with platform issues and coding bugs (PDF&PSI&CIB).

5.2. Crypto APIs and cryptography learning

In the previous study, we observed that repeated measures were made for some developers. This fact motivated us to perform a retrospective, longitudinal study [49] to analyze developers behavior from a series of observations already made about them. We found that developers (which were active users of those two communities) not only failed in giving good answers to questions related to cryptography; sometimes, they also omitted information that could prevent cryptography misuse.

In OJC, we observed a notable presence of misuses CAI, PKC, and WC. Also, the numbers for simple misuses (WC and CIB) were relatively stable (not decreasing) over time, suggesting that simple misuses were recurrent and developers were not getting better at them. Also, the number of CAI issues decreased over time, suggesting an increase in knowledge about Java's crypto API.

In GAD, we saw a notable decrease in crypto misuse early in developer's lifespan, suggesting that developers had a fast learning curve for those misuses less influenced by platform specific issues. Also, four misuse categories (WC, CIB, PKC, and IVM) were recurrent during developers' lifespans, suggesting that Android's diverse ecosystem difficulties learning cryptography for both simple and complex misuses.

A straightforward analysis on misuse density over time was used to evidence a learning curve for developers as well as to show misuse reduction (or growing). In OJC, we observed that misuse density was relatively stable over time, despite a gradual reduction in density for platform issues. Also, we observed that density of simple misuses (WC, CIB, and BR) increase over time, while density of moderate misuses (PDF, PKC, and ICV) had a small decrease, and density for high-complexity misuses (CAI, PKM, and IVM) showed a gradual decrease.

Our observations suggested that simple misuses were recurrent in OJC, not depending on the actual knowledge of Java's crypto API. On the other hand, medium-complexity misuses were the most influenced by platform issues. Then, complex misuses decreases over time, suggesting a gradual improvement in developer's knowledge about Java's crypto API.

Similarly, in GAD, we observed that misuse density increased over time. Also, density for complex misuses was relatively stable, while density for low- and medium-complexity misuses grown over time. This behavior suggested that Android developers were not getting better in cryptography over time.

5.3. Discussion of empirical studies

We found that inherently complex, hard-to-use architectures distract developers from actual cryptography misuse and contribute to perpetuate recurring errors in cryptographic programming. Also, developers were not aware of design flaws and take for granted parameters generated by tools.

Security is a secondary concern for developers, which usually have priorities (e.g., functional correctness, time to market, maintainability, compliance) that often conflict with security. Frequently, developers look for quick, but insecure solutions and online communities favor this behavior.

Ideally, developers should not be forced to learn cryptography in order to correctly use crypto APIs, specially for simple use cases. However, in practice, crypto APIs are unable to foster their correct use without domain knowledge obtained from elsewhere but online communities.

Developers learn how to make APIs work, but this does not mean cryptography was used correctly. In fact, coding bugs are persistent issues when using general-purpose (function-based) crypto APIs to implement application-specific use cases, because developers are forced to make insecure choices without actually understanding the whole situation.

6. Static analysis tools and cryptography misuse

As discussed, static code analysis tools (SCATs) are considered by software security communities the most cost-effective way to find security bugs in early stages of software development. We evaluated [50] five free SCATs in order to find out how and to which extent cryptography misuse can be detected by free SCATs currently available to developers. We choose to evaluated free SCATs because these tools are readily available (no purchase of licences is required) and, in many cases, are the first and only option for ordinary developers.

We exercised SCATs using test cases for cryptography usage (with and without crypto misuses) and, based on detected misuses, made measures (TP, TN, FN, and FP) and calculated metrics (e.g., precision, recall, and f-measure) that captured the detection capabilities of SCATs in the cryptography domain. Test cases were derived directly from our classification of cryptography misuse by writing programs to exemplify misuse instances and their variants.

We selected five free SCATs for Java from OWASP [86] and SAMATE [87]: FindBugs 3.0.1 [88] with FindSecBugs 1.5.0 [89], VisualCodeGrepper 2.1.0 [90] (VCG), Xanitizer 3.0.0 [91], SonarQube 6.2 [92] with sonar-scanner 2.8, and Yasca 3.0.5 [93]. All these tools perform late detection of vulnerabilities [94] and should be applied after developers have produced some source code.

We found that all evaluated SCATs preferred misuses from MG1 and gradually decreased their performance for MG2 and MG3. We also found that, in general, tools perform better in simple misuses regarding weak cryptography (WC) and bad randomness (BR), and worse in issues for key management (PKM) and program design flaws (PDF).

Most evaluated tools used pattern matching as the main technique for vulnerability detection. In particular, VCG, SQ, and Yasca used only simple pattern matching. FSB detected a few sophisticated patterns, while Xan used taint analysis to detect simple leaks of keys and indirect references to weak algorithms. Also, FSB and Xan reported all occurrences of encryption modes that turned block ciphers into stream ciphers (e.g. OFB and CFB),

whether they were used correctly or not, showing a lack of discrimination [95].

Tools were not mutually exclusive and had great intersection. When computing the union of TPs for all tools, we found a general coverage of around 35%. That is, the union of misuses detected by all five tools covered only about 35% of crypto misuses in our test cases. The tool with higher recall detected only one third (around 33%) of all misuses. The second higher recall detected one quarter (25%) of misuses. These numbers were not surprising. For instance, other benchmarks [96–98] of SCATs found a coverage of 50% for the best tool. These results suggested that tools perform better in other security domains than in cryptography.

Optimistic tools only alert about clear misuses and keeps silent about dubious ones, while pessimistic tools warn about every suspected misuse, even unlikely ones. SCATs for security are pessimistic in order to avoid dangerous omissions [25]. Also, SCATs should favor early detection of vulnerabilities in order to benefit from developer's short-term memories when fixing vulnerabilities [94].

Based upon our findings, we generalized the behavior for crypto-friendly SCATs for those misuse groups related to coding (MG1), design (MG2), and architecture (MG3), described in Section 4.2. First, in MG1 (WC, CIB, and BR), tools can be quite precise in early detection of crypto misuses, showing relatively few FPs. Recall inside the misuse group (group recall) is expected to be relatively high, with few FNs. However, non-detected misuses from MG2 and MG3 cause dangerous omissions (FNs in overall recall).

Second, in MG2 (ICV, PDF, and PKC), tools are expected to be less precise, producing more false alarms and omissions than in MG1. This is expected due to FPs caused by incomplete understanding of program design, as well as FNs due to misconceptions about programs. In MG2, optimistic tools are expected to show relatively low recall and many FNs, while pessimistic tools are expected to have relatively low precision and many FPs. These tools are better suited to late detection of crypto misuses.

Third, in MG3 (PKM, IVM, and CAI), tools are expected to be quite imprecise, producing more false alarms and omissions than in other misuse groups. This behaviour is expected because of FPs due to partial understanding of software architectures, as well as FNs due to misconceptions about program design. Optimistic tools will have relatively low recall, while pessimistic tools will have lower precision. Tools for MG3 are better suited to late detection of misuses.

6.1. Cryptography misuse in use cases and coding tasks

The application domain defines the cryptography requirements [99], which, in general, are satisfied by (but not limited to) traditional use cases associated to cryptographic services [48].

We found that there are simple use cases associated to cryptographic services [48], which are easily recognized by developers: Encrypting Data at Rest (EDR), Secure Communication (SC), Password Protection and Encryption (PPE), and Authentication and Validation of Data (AVD). This list is not exhaustive and can be augmented by sophisticated use cases where cryptographic services can be blended to application functionality in novel ways, as we identified in selected development cases (from Section 2.2). For instance, we found that Android developers have a specific use case for cryptography regarding Digital rights Management (DRM).

These use cases, either simple or complex, are implemented by coding tasks [48], also easily recognized by developers in a non-exhaustive list: encryption and decryption (Enc/Dec), digital signatures and verification (Sig/Ver), hashes or authentication codes and verification (Hash/Mac), key generation or agreement (KA), secure channels (e.g., SSL/TLS), digital certificate validation (Cert), and randomness generation (Rand). There are also secondary tasks, accessory to the main ones, representing operational (but important) aspects of crypto systems: key distribution, certificate generation, and key storage and recovery. In general, every crypto use case can be accomplished by a combination of coding tasks, where the complexity of the task is determined by the actual use case at hand.

Cryptography misuse is introduced by developers into use cases during coding tasks [48]. As discussed before, cryptography misuse instances are not all equally difficult to avoid: some are easier to find and correct than others, depending on the level of abstraction (e.g., code, design, and architecture) required to identify the misuse [48, 50], as detailed in Section 4.2, in three groupings of the nine original misuse categories: MG1 (WC, CIB, and BR), MG2 (PDF, ICV, PKC), and MG3 (PKM, IVM, and CAI).

This section correlates our empirical studies [48, 49] on cryptography misuse with the Java's crypto API [82], and the experimental evaluation of static analysis tools [50], associating them to use cases and coding tasks for cryptography.

The remaining of this subsection is organized as follows. Subsubsection 6.1.1 details the relation between crypto misuses and use cases for cryptography. Subsubsection 6.1.2 details the relation between crypto misuse and coding tasks for cryptography. Subssubection 6.1.3 gives conclusions about these relations.

6.1.1. Cryptography misuse and use cases

Concerning those use cases associated to cryptography from [48], we learned that the most-frequent use case (in those online communities targeted by our studies) is Encrypting Data at Rest (EDR), showing high percentegaes in both OJC (31%) and GAD (65%). Authentication and Validation of Data (AVD) is the second most frequent in OJC (26.5%) and third in GAD (8.5%). Secure Communication (SC) is third most frequent in OJC (18%) and second in GAD (11%). Password Protection and Encryption (PPE) is frequent in GAD (11%). Table 3 summarizes these numbers and adds how static code analysis tools (SCATs) detect crypto misuse by use cases, considering only a raw measure of all detected misuses (by all evaluated tools) in our test cases.

These use cases suffer from crypto misuse as follows. In OJC, most misuse were associated to encrypting data at rest (EDR), with attention to two misuse categories: weak cryptography (WC) and coding bugs (CIB). Also, Secure communication (SC) is affected by mistakes in certificate validation and (weak) public-key cryptography. Authenticating or validating data (AVD) is affected by coding mistakes (CIB) specific to Java. In GAD, most misuse were associated to encrypting data at rest (EDR) with special attention to design flaws and IV/nonce management (IVM). Also, Secure communication (SC), password protection (PPE), and authentication and validation of data (AVD) are moderately affected by misuses. Secure communication (SC) is negatively affected by complexity of Android's certificate storage.

When evaluating static analysis tools for cryptography [50], we learned that, besides an overall coverage of about 35%, tools behave differently for different crypto use cases. Two use cases, EDR e AVD, have a coverage around 40%, while RND reaches 46%. Two use cases are less covered by tools, PPE with 30% and SC with 18%. Table 3 summarizes these numbers and relates them to crypto misuse in use cases from online communities.

Interestingly, the two most-frequent use cases (EDR and AVD), which were associated to simple misuses (WC and CIB), also had high coverage of misuses by tools, suggesting that tools favor frequent use cases with simple misuses.

In spite of being moderately frequent use cases, SC and PPE showed a low coverage of misuse by tools, suggesting that tools do not favor secure communication or encryption for password protection, because these use cases are affected by complex misuses for public-key cryptography (PKC) issues and improper certificate validation (ICV), where tools had many omissions.

The high percentage for detection of misuses associated to PRNGs suggests that tools can solve many simple misuses of random numbers in cryptography. In fact, this kind of misuse did not show up in the study of online communities, suggesting that simple instances of this misuse category were not common in developers discussions, but have hidden pitfalls not recognized by ordinary developers nor tools.

6.1.2. Cryptography misuse in coding tasks

Concerning coding tasks associated to cryptography from [48], we learned that the most-frequent coding task is Encryption/Decryption, in both OJC (26%) and GAD (32%). Digital certification (17% in OJC and 1.5% in GAD), as well as Signatures and hashes/MACs (16.5% in OJC and 10% in GAD) are moderately frequent. SSL/TLS secure channel is the less-frequent coding task (8.5% in OJC and 1.5% in GAD). Table 4 summarizes the occurrence of crypto misuse for these coding tasks for cryptography, comparing them to the performance of static analysis tools. Again, considering only a raw measure of all detected misuses (by all evaluated tools) in our test cases.

In OJC, most misuses were associated to coding tasks for encryption (Enc), followed by key generation (KG) and SSL secure channels. Bad randomness (BR) did not show up in any programming task for this community. In GAD, most misuse affected the coding task for encryption. Also, weak cryptography and coding bugs were the most perceived crypto misuses. Signing was moderately affected by many crypto misuses. Key generation and certificate handling were tasks scarcely affected by crypto

Table 3: Cryptography misuse and tool support by use case.

	Use cases				
	EDR	SC	PPE	AVD	RND
OJC	31%	18%	0%	26.5%	-
GAD	65%	11%	11%	8.5%	-
SCATs	0.4	0.18	0.3	0.408	0.46

Table 4: Cryptography misuse and tool support by coding task.

	Coding Tasks					
	Enc	Sig & Hash/MAC	KA	SSL	Cert	Rand
OJC	26%	16.5%	11%	8.5%	17%	–
GAD	32%	10%	4%	1.5%	1.5%	–
SCATs	0.398	0.461 & 0.273	0.256	0.4	0.0	0.5

misuse. Programming of SSL channels did not show any relevant crypto misuse in Android, suggesting that GAD developers did not know what to ask regarding crypto misuse in these programming tasks.

Table 4 shows that tools also behave differently for distinct coding tasks for cryptography. Two tasks have relatively high coverage: Rand (50%) and Sign (46%). Two tasks have moderate coverage: Enc (39%) and SSL (40%). Other two tasks have relatively low coverage: KA (27%) and Hahs/MAC (27%). Cert was not covered at all.

Coding tasks for encryption were the most common in communities, but only moderately covered by tools. Programming of SSL/TLS channels had a relatively good coverage by tools, despite being not so popular in communities. Coding for PRNGs had a relatively good coverage by tools, despite not being mentioned in OJC.

Misuses in Encryption/Decryption and Signatures, hashes/MACs are relatively high covered by tools, suggesting that (again) tools favor detection of misuses in frequent coding tasks with simple misuses. Evaluated tools neglected misuses of digital certification, despite this coding task being frequent.

6.1.3. Correlating cryptography misuse by developers and tool's support to cryptography

In summary, we found that static analysis tools favor the detection of simple crypto misuse within most frequent use cases and coding tasks, suggesting a prioritization of efforts by tool builders in order to offer a market trade-off between supporting simple misuses in frequent cases and tasks against neglecting sophisticate misuses and rare use cases and tasks.

Finally, when we correlate the occurrence of cryptography misuse in online communities (OJC and GAD) to the evaluation of tools with the f-measure metric (in Table 5) for the two static analysis tools best ranked in our experiment [50], we confirm that, in general, simple misuses of cryptography, those ones related to code, are both common in coding communities and preferred by static analysis tools.

We also conclude that, considering the current state of maturity in understanding of how cryptography misuse manifest itself in software, both developers and tool builders perceive more clearly almost only low-complexity misuse instances, those ones related to code-level vulnerabilities in simple use cases.

7. A Methodology for Developing Secure Cryptographic Software

This section revisits our methodology for development of cryptographic software in order to enhance it with new findings regarding cryptography misuse.

Over the years, we have observed the frequent adoption of common practices by researchers and practitioners, which gave us the required perspective to generalize a working methodology for development of secure cryptographic software. This section presents this methodology as an ordered way to approach cryptography into Secure Software Development Life Cycles (SSDLC).

Our methodology, named Development of Secure Cryptographic Software (DSCS) [99], is part of the knowledge base for Cryptographic Software Security and emerged as a response to practical needs in building cryptographic software observed when we proposed design patterns for cryptography [100], built cryptographically secure mobile

Table 5: Cryptography misuse correlated to tool support with f-measure metric.

	MG1			MG2			MG3		
	WC	CIB	BR	PDF	PKC	ICV	PKM	IVM	CAI
OJC	26%	17%	0%	6%	16%	4%	11%	5%	20%
GAD	21%	17%	1%	8%	10%	3%	4%	6%	1%
Xan	0.541	0.571	0.588	0.270	0.381	0.235	0.263	0.286	0.889
FSB	0,516	0,256	0,588	0,286	0,361	0,421	0,263	0,364	0,000

apps [3–6], applied Acceptance Test Driven Development (ATDD) to cryptographic services [101] (when building [102] and porting [103] a cryptography library to mobile devices), surveyed tools for secure programming and verification of cryptographic software [33], trained developers in cryptography programming [75], investigated crypto misuses in online communities [48, 49], and evaluated static code analysis tools for cryptograhy [50].

Now, the contribution of this text to the field of cryptographic software security is twofold. First, to explicitly incorporate a validated classification of cryptography misuse for software security into the methodology. Then, to give stronger evidence of cryptography misuse by developers and tools, in order to better support the use of this methodology in secure software developments.

This section is organized as follows. Subsection 7.1 details the steps for conducting DSCS. Subsection 7.2 explains how to foster a layered architecture for cryptographic software. Subsection 7.3 analyses tool support for DSCS with a particular attention to static code analysis tools. Subsection 7.4 discusses the integration of DSCS into SSDLC methods.

7.1. Steps for DSCS

DSCS comprises a set of steps related to SSDLC phases, resembling software engineering methods, and an extra phase for deployment: requirements, design, construction, testing, and deployment and operation. The following paragraphs describe the steps.

Step 1: Crypto software requirements. Security policies, regulations, business needs, non-functional requirements, and predefined checklists determine which security goals must be accomplished. Security goals (e.g., anonymity, confidentiality, integrity, non repudiation, authenticity, etc.) are then mapped to predefined cryptographic features. These mappings can follow both common **use cases** or user stories and developer's **coding tasks**, facilitating the detection and avoidance of likely crypto misuses. Potential crypto misuses from MG3 related to architectural choices (e.g., crypto agility issues, PKI issues, multiple access points, and randomness sources) and key management (e.g., PBE passwords, expired keys, key distribution, and CA issues) can be identified and avoided in this step. Also, the definition of policies and checklists help to avoid misuses in MG1 and MG2 at later phases. This step is usually supported by software security practitioners or security architects. Crypto experts support the specification of new user stories (use cases) as well as the related programming tasks and crypto features.

Step 2: Crypto software design. Cryptographic features, design goals, and predefined control types contribute to define a security architecture, which contains cryptographic services and technologies. In this step, cryptographic design patterns can provide proven solutions to common use cases and coding tasks traditionally associated to cryptography. Some crypto misuses from MG3 (e.g., IV/nonce management issues, API misunderstandings, improper key length, and reused keys), many misuses from MG2 (e.g., design flaws, certificate validation issues, and PKC issues), and a few misuses from MG1 (e.g, proprietary cryptography, custom implementation, and risky/broken cryptography) can be identified and avoided in this step. This step is usually supported by security architects or software security practitioners. But, cryptography experts may provide timely advice.

Step 3: Crypto software construction. This step consists in programming secure crypto software. Code is written in coding tasks to integrate crypto controls and features into software functionality, according to use cases or user stories. The source code must follow conventions of crypto APIs and adopts standard implementations of algorithms and protocols, which are offered by frameworks and reusable libraries. Many crypto misuses from MG2 and all misuses from MG1 can be identified and avoided in this step. The work of software security practitioners receives most support from experts in

applied cryptography in order to avoid crypto misuses. In this step, the proper use of static analysis tools, either directly by developers or in supporting reviews by experts, can benefit any development team.

Step 4: Crypto software testing. Cryptography-related functionality and packages are submitted to security tests of two types: functional security tests (supported by security-inspired test cases) and penetration tests, supported by attack scenarios and threats. This step includes tests for misuses from MG2 and MG1, as well as security verification (e.g., inspections) of misuses from MG3. In this step, the work of software security practitioners is supported by experts in cracking crypto software, usually by exploiting remaining crypto misuses and other implementation bugs.

Step 5: Crypto software deployment and operation. In this step, secure software is continuously monitored and periodically tested for violations of policies and guidelines, as well as for discovery of new vulnerabilities. This step is usually performed by a software security practitioner. Continuous attention should be given to the fast and constant evolution of cryptographic technology in terms of new standardized algorithms, updated best practices, and adoption of longer key lengths. This step includes tests for misuses from MG2 and MG1 as well as assessments for misuses from MG3.

The above steps are quite straightforward and capture the common practices we have seen so far. They may not be the best choices made by practitioners, but the possible ones due to several constraints. For instance, it is quite common to involve cryptography experts only in later steps of SSDLC.

7.2. A reference architecture for cryptographic software

Modern crypto software has to follow software engineering best practices. In previous works we proposed design patterns for cryptographic systems [100] and a layered architecture for cryptographic software [99]. These layers have to work together to promote the secure use of cryptography, and constitute the cryptographic software stack, as follows:

1. **The user interaction layer**. This layer has to be able to promote the proper use of cryptography, preventing users from misusing cryptography, and transparently blending security-sensitive functionality to cryptographic features. Today, it is well accepted [104**?** –108] that ordinary users have great trouble in using cryptosystems by themselves and should not have direct access to cryptosystems' operations. Thus, user error in operating cryptosystems should be avoided by making cryptography transparent or invisible to final users.
2. **The business logic layer**. This layer should be able to properly orchestrate crypto services and components, with adequate use cases and designs expected by developers when securing sensitive business goals, avoiding crypto misuses from MG3 and MG2. This layer is where cryptography can be blended to application functionality in unanticipated ways.
3. **APIs and frameworks layer**. This layer must be able to provide access to cryptographic implementations in standardized and decoupled ways, so that both the replacement and the exchange of implementations are easily achieved, avoiding crypto misuses from MG2 and MG1. Also, APIs, libraries, and frameworks should not disclose sensitive information through unauthorized side channels;
4. **Algorithm and protocol layer**. In this layer, cryptographic implementations of algorithms and protocols should be robust against various failures (e.g. hardware and memory failures), secure against various attacks (timing, side-channel leakages, etc.), efficient in energy (low power consumption) and computational resources (CPU cycles, memory, etc.), and compact for use in restricted environments;
5. **Development support and infrastructure layer**. In this layer, programming languages, component libraries, compilers, obfuscators, security tools, and even operating systems should be able to capture the programmer's intent, detecting deviations, preserving security decisions, and not canceling protections when translating source or binary code of cryptographic implementations to machine code.

Additionally, the mathematical security of cryptographic algorithms and protocols is, in general, an assumption supported by choosing good-standing standards as well as cryptography of good reputation.

These layers can be associated to DSCS steps, in specific instantiations. For instance, the user inter-

action layer may relate to requirements elicitation and business modeling (software requirements security); the business logic layer may relate to design and architecture (software design security); the API and frameworks layer may relate to secure coding and construction (software construction security); and security of crypto components is related to verification and validation against predefined requirements and new threats (software testing security).

Finally, we suggest a new role in software development capable of handling the security issues related to the architecture of cryptographic software: the **cryptographic architect**. Unlike the cryptographic engineer, the cryptographic architect is not primarily concerned with secure construction of cryptographic functions, neither with internal workings of mathematical structures, because for him, the quality of an algorithm's implementation is taken for granted by adopting good-stading resources obtained from a trusted cryptographic infrastructure.

The cryptographic architect is much rather interested in safe composition and use of cryptographic components, possibly at different levels of software abstraction. He puts great emphasis on the cryptographic architecture of software applications, with an interest on how it can be securely reused by other software, considering not only the (social) context in which cryptography is applied, but also the forces influencing its implementation.

In multidisciplinary software systems, like modern cryptographic software, differences in disciplines can cause crypto misuse. For instance, the involvement of a range of professional expertise is essential because there are other aspects to be tackled, different from cryptograhy and directly related to business logic as well as other non-functional requirements. However, differences among disciplines can introduce cryptography misuse and compromise the security of the software being developed.

This may happen because each involved discipline makes assumptions about what can be done by other disciplines, based on inadequate understandings. Disciplines try to protect their boundaries and may argue for certain design decisions because these decisions will call for their expertise. This is why an explicit role for a crypto architect is necessary to protect cryptography from being misused by software developers and negatively influenced by design decisions.

7.3. Analysis of tool support for DSCS

Currently, the development of secure crypto software has quite limited tool support for most issues. The coverage of cryptography misuse by tools is far from good, with current tools showing many blind spots and a huge gap between what experts actually see as cryptography misuse and what tools can detect [50]. Therefore, expert help is required to assure quality in different moments of development efforts. We argue that, with current tools' maturity, an adequate toolkit has to be carefully crafted to fit the needs of specific development contexts.

The development of proprietary cryptography (proprietary algorithms or homemade implementations) by ordinary programmers is, in general, considered bad practice. Therefore, tools for secure programming of cryptographic algorithms and libraries (i.e., tools applied below crypto APIs) are omitted from this toolkit. A previous work [33] studied tools for secure coding and testing of crypto software. Here, we focus on tools for development of cryptography-enabled functionality with established and standardized algorithms, as well as trusted libraries and frameworks (i.e., tools applied above crypto APIs).

7.3.1. Assembling a toolkit for DSCS

This section is an attempt to assemble a hypothetical toolkit for DSCS, which is based only on related works. This toolkit may include tools for both secure programming and verification. For crypto software construction, SSL/TLS frameworks [83] offer high-level functionality and configurable services, instead of primitive functions, enhancing usability and avoiding unintentional mistakes. Also, specialized tools, such as OpenCCE [109], can be integrated to IDEs and guide developers through selection and use of relevant crypto features for use cases or coding tasks, automatically generating code with suitable API calls, avoiding mistakes by non-experts.

For those use cases not supported by automatic generation of code, current coding standards [69, 72] do offer simple rules for cryptography misuse that can be automated by simple static analysis tools (e.g., FindSecBugs [89], SonarQube [92], Xanitizer [91], VisualCodeGrepper [90], and Yasca [93]).

However, sophisticated issues cannot be detected by ordinary tools and have only been addressed by prototypes. For instance, CryptoLint [36] and

Cryptography Misuse Analyzer (CMA) [37], for Android, and ICryptoTracer [110], for iOS, are static analysis tools that identify predefined sets of misuses and vulnerabilities from API calls. Also, both SSLint [111] and MalloDroid [40] are static analysis tools for detecting incorrect use of SSL/TLS APIs and improper certificate validation, detecting potential man-in-the-middle attacks.

For crypto software testing, tests for SSL have been used for detection of HTTPS misconfigurations in web apps [30]. Also, MalloDroid [40] can be used to dynamically detect SSL vulnerabilities against Android apps. The Padding Oracle Exploitation Tool (POET) [42] automatically finds and exploits this type of side channel. Also, Fault Injection Attack Tool (FIAT) [112] can inject malicious faults into cryptographic devices.

Finally, experience shows that tools are incomplete, not overlapping, and buggy. Thus, various tools should be combined to obtain diversity and redundancy, promoting fault tolerance to DSCS. In testing, the incompleteness and absence of tools is more evident than in construction. Therefore, household tool development and customization, as well as manual testing skills, have been favored in practice.

7.3.2. Static code analysis tools for cryptography

As discussed in Section 6, tools can be quite precise in early detection of MG1 misuses. However, non-detected misuses from MG2 and MG3 can cause dangerous omissions. For MG2, tools are expected to be less precise, producing more false alarms and omissions than for MG1. These tools are better suited to late detection of crypto misuses. For MG3, tools are expected to be quite imprecise and are better suited to late detection of misuses. Based upon our observations [50], we generalized the following expected usages of SCATs in DSCS.

By precisely detecting many code-based misuses in MG1, SCATs can be integrated to IDEs and provide real-time support to developers during coding tasks (e.g., early detection). Since precision is high, these tools will need less expert supervision in coding. Because overall recall is low (only MG1), tools alone will result in low quality (less secure) software, which can be subjected to further verification.

SCATs should not be the only way to find misuses in MG2 (e.g., design flaws in coding), because these misuses are supposed to be found earlier in development, before coding tasks. Tools can be applied during system integration (daily or weekly build) and support design reviews or manual inspection (by experts), possibly later in system development.

Similarly, for detecting misuses in MG3, SCATs should not be the only way to find architectural flaws in coding, because they are supposed to be found very early in development. Extensive use of SCATs during coding tasks is inadequate and can mistakenly divert team effort to correct nonexistent misuses or correct existing, but complex, misuses the wrong way. This tool usage is better for supporting experts during manual inspections in architectural security analysis.

7.4. Integrating DSCS into SSDLC

At Section 3.5, Table 1 qualitatively compared main SSDLC approaches or schools for schoftware security. We observed that software security schools have no specific activities for cryptography. However, the development of crypto software is influenced by the dominant school, which determines the way cryptography is approached. We argue that SSDLC methods can be adapted to empower developers with proper ways to avoid cryptography misuse when building crypto software.

For instance, when practitioners show an inclination to a process-driven approach, preferring the predictability of a waterfall-like SSDLC (which is driven by known requirements, resulting in software able to pass well-defined evaluation criteria), our classification of cryptography misuse provides a practical checklist to watch and follow during development.

Other practitioners may prefer a technology-driven approach, arguing that cryptographic features are mostly well-defined controls that will be put in place anyway, with no need for deep threat modeling or traceable processes. For them, coding guidelines supported by our classification and a good (secure enough) architecture are sufficient to securely apply cryptography. In this case, there is no particular interest in applying fine-grained risk analysis deeply into daily activities for crypto software, leaving risk analysis at the business level.

Also, modern software development frequently adopts highly adaptive life cycles [28], characterized by progressive specification of requirements based on short iterative development cycles, where processes are not rigid paths, risk analysis is fast enough to not slow down the team, and risk gradually reduced over time by the evolution of understanding.

Cryptographic software is frequently multidisciplinary and, when developed according to highly adaptive life cycles, involves many related entities and has no definitive problem statement or requirements specification until the work is done. Different stakeholders see the software being built in different ways and no one has a full understanding of the system as a whole. In extreme cases, the true nature of the software may only emerge as a solution is developed.

In this case, a technology-driven DSCS is a better fit to steadily evolve understanding from crypto use cases, to coding tasks, to the avoidable crypto misuses in our classification. In general, a technology-driven DSCS can be adopted bottom-up and conducted by technical staff (supported by executives). For instance, a lower manager can enhance the software security capabilities of his staff by integrating into the SSDLC a well-crafted set of tools and techniques for mitigating cryptography misuse.

8. Broad discussion

This section discusses those points common to all aspects of the broad investigation. Other specific topics were discussed locally, in their corresponding section or subsection.

Historically, the responsibility for the correct use (or misuse) of cryptography has been attributed, simply, to the software developer. We showed that this attribution is unfair, since the causes of security issues found in cryptographic software are not just the careless (or malicious) programmer, but, rather, are associated to the paradigm currently adopted in the development of cryptographic software.

By correlating findings from several facets of cryptographic software development (namely, static analysis tools, software security textbooks, coding communities, and research on cryptography misuse), we observed that developers misuse APIs due to their lack of knowledge in cryptography as well as excessive complexity of APIs, while security experts build flawed test cases for detecting cryptography misuse (e.g., in static code analysis tools) due to lack of knowledge in coding as well as in system design and architecture.

On the other hand, the development of crypto software suffers great influence from the supporting SSDLC, as well as from social groups, or communities, which ultimately determine the way cryptography is approached in development of secure software. We argue that SSDLC methods can be adapted to empower developers with proper ways to avoid cryptography misuse when building crypto software.

When investigating developers using crypto APIs as well as static analysis tools built by security experts, we noticed almost the same knowledge gaps and an overlap of blind spots. For instance, coding bugs are persistent issues when using general-purpose (function-based) crypto APIs to implement application-specific use cases, because developers are forced to make insecure choices without actually understanding the whole situation, due to many factors. In one hand, quick advice given in online communities are shallow and incorrect in cryptography matters and literature is too arcane or obsolete. On the other hand, tools are incomplete, non-overlapping and buggy, and experts in applied cryptography are barely available. We found that both builders of static analysis tools and experts in software security underestimate cryptography misuse. Therefore, both security tools and security textbooks overlook cryptography matters.

By comparing coding communities and textbooks we observed that textbooks (when updated) are supposed to offer better ways to obtain knowledge on cryptography. Online communities are better for timely advice in troubleshooting technology-specific issues. Ideally, developers should not be forced to learn cryptography in order to correctly use cryptographic APIs, specially for simple use cases. However, in practice, crypto APIs are unable to foster their correct use without domain knowledge obtained from elsewhere but online communities.

Concerning the use of tools, currently, the development of secure crypto software has quite limited tool support for most issues. The coverage of cryptography misuse by tools is far from good, with current tools showing many blind spots and a huge knowledge gap between what experts actually see as cryptography misuse and what tools can detect. Therefore, expert help is required to assure quality in different moments of development efforts. We argue that, with current tools' maturity, an adequate toolkit has to be carefully crafted to fit the needs of specific development contexts.

There is a profile of cryptography misuse for each application enabled by cryptography. The development profile of a cryptographic software reflects the likely crypto misuses developers do in practice. The likelihood of a particular crypto misuse depends on the use cases and related coding tasks to accomplish them. By knowing the use cases and coding tasks

for a cryptographic software, its is possible to anticipate the likely crypto misuses, and better direct efforts to mitigate and avoid them, in an structured way, according to our methodology for development of secure cryptographic software.

9. Concluding remarks

This text suggests that Cryptographic Software Security is a new field of study for developing secure cryptographic software. The body of knowledge supporting this field of study emerged from literature reviews (mapping studies) and development cases; was validated by empirical studies and experimental analysis; and provides a validated classification of cryptography misuse for software security and a working methodology for development of secure cryptographic software.

We foresee, in the near future, the emergence of a new generation of tools (e.g., developer-friendly APIs, programming frameworks, code generators, and static code analysis tools) to better assist software developers in programming cryptographic software, as well as new ways to instruct developers on how not to use cryptography. In the long run, further investigation will uncover other architectural aspects of cryptographic software, broadening the understanding about design flaws in cryptography and opening opportunities for effective architectural frameworks.

Acknowledgements

Alexandre Braga thanks CNPq and Intel for the financial support, as well as the University of Campinas and CPqD for the institutional support. Ricardo Dahab thanks FAPESP, CNPq, CAPES, and Intel for partially supporting this work.

10. References

[1] M. U. A. Khan, M. Zulkernine, On Selecting Appropriate Development Processes and Requirements Engineering Methods for Secure Software, 33rd Annual IEEE International Computer Software and Applications Conference (2009) 353–358doi:10.1109/COMPSAC.2009.206.

[2] I. Sommerville, Software Engineering, International Computer Science Series, Pearson, 2011.

[3] A. Braga, D. Schwab, Design Issues in the Construction of a Cryptographically Secure Instant Message Service for Android Smartphones, in: The 8th Inter. Conf. on Emerging Security Information, Systems and Technologies, 2014, pp. 7–13.

[4] A. Braga, R. Zanco Neto, A. Vannucci, R. Hiramatsu, Implementation Issues in the Construction of an Application Framework for Secure SMS Messages on Android Smartphones, in: The 9th Inter. Conf. on Emerging Security Information, Systems and Technologies, IARIA, 2015, pp. 67–73.

[5] A. Braga, A. Colito, Adding Secure Deletion to an Encrypted File System on Android Smartphones, in: The 8th Inter. Conf. on Emerging Security Information, Systems and Technologies, 2014, pp. 106–110.

[6] A. Braga, D. Schwab, E. Morais, R. Neto, A. Vannucci, Integrated Technologies for Communication Security and Secure Deletion on Android Smartphones, International Journal On Advances in Security 8 (1&2) (2015) 28–47.

[7] Skillsoft Books24x7.
URL `https://acm.skillport.com`

[8] O'Reilly Safari Books Online.
URL `https://www.safaribooksonline.com`

[9] A. K. Talukder, M. Chaitanya, Architecting secure software systems, CRC Press, 2008.

[10] J. Ransome, A. Misra, Core Software Security: Security at the Source, CRC Press, 2013.

[11] A. Shostack, Threat modeling: Designing for security, John Wiley & Sons, 2014.

[12] M. S. Merkow, L. Raghavan, Secure and Resilient Software Development, CRC Press, 2010.

[13] M. Howard, D. LeBlanc, J. Viega, 24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them, McGraw-Hill Education, 2009.

[14] C. Douglas A. Ashbaugh, Security Software Development: Assessing and Managing Security Risks, Taylor & Francis, 2008.

[15] S. Harris, CISSP All-in-One Exam Guide, 6th Edition, All-in-One, McGraw-Hill Education, 2012.

[16] R. Anderson, Security engineering, 2008.

[17] T. Richardson, C. Thies, Secure Software Design, Jones & Bartlett Publishers, 2012.

[18] J. Viega, G. McGraw, Building Secure Software: How to Avoid Security Problems the Right Way, 2001.

[19] M. Dowd, J. McDonald, J. Schuh, The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities, Pearson Education, 2006.

[20] G. McGraw, Software Security: Building Security in, 2006.

[21] J. H. Allen, S. Barnum, R. J. Ellison, G. McGraw, N. R. Mead, Software Security Engineering: A Guide for Project Managers (The SEI Series in Software Engineering), 1st Edition, SEI Series in Software Engineering, Addison-Wesley Professional, 2008.

[22] M. Howard, D. LeBlanc, Writing secure code, 2003.

[23] M. Howard, S. Lipner, The Security Development Lifecycle, Microsoft Press, Redmond, WA, USA, 2006.

[24] M. Paul, Official (ISC)2 Guide to the CSSLP, (ISC)2 Press, Taylor & Francis, 2011.

[25] B. Chess, J. West, Secure programming with static analysis, 2007.

[26] N. Daswani, C. Kern, A. Kesavan, Foundations of Security: What Every Programmer Needs to Know, Apress, Berkely, CA, USA, 2007.

[27] G. Hoglund, G. McGraw, Exploiting software: How to break code, Pearson Education India, 2004.

[28] PMI, Software Extension to the PMBOK® Guide, 5th Edition, Project Management Institute, 2013.

[29] P. Bourque, R. Fairley (Eds.), Guide to the Software

Engineering Body of Knowledge (SWEBOK), version 3. Edition, IEEE Computer Society, 2014.
[30] OWASP, OWASP Testing Project (2015). URL https://www.owasp.org/index.php/OWASP_Testing_Project
[31] N. Husted, S. Myers, Emergent Properties and Security: The Complexity of Security As a Science, in: Proceedings of the 2014 Workshop on New Security Paradigms Workshop, NSPW '14, ACM, New York, NY, USA, 2014, pp. 1–14. doi:10.1145/2683467.2683468.
[32] N. Leveson, Engineering a Safer World Systems Thinking Applied to Safety, MIT Press, 2011.
[33] A. Braga, R. Dahab, A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software, in: XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg 2015), Florianópolis, SC, Brazil, 2015, pp. 30–43.
[34] Codenomecon, The Heartbleed Bug (2014). URL http://heartbleed.com/
[35] A. Langley, Apple's SSL/TLS "Goto fail" bug (2014). URL www.imperialviolet.org/2014/02/22/applebug.html
[36] M. Egele, D. Brumley, Y. Fratantonio, C. Kruegel, An empirical study of cryptographic misuse in android applications, ACM SIGSAC conference on Computer & comm. security (CCS'13) (2013) 73–84doi:10.1145/2508859.2516693.
[37] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, S. Chenjie, Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications, in: IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC), 2014, pp. 75–80. doi:10.1109/DASC.2014.22.
[38] D. Lazar, H. Chen, X. Wang, N. Zeldovich, Why Does Cryptographic Software Fail?: A Case Study and Open Problems, in: 5th Asia-Pacific Workshop on Systems, APSys '14, ACM, New York, NY, USA, 2014, pp. 7:1—-7:7. doi:10.1145/2637166.2637237.
[39] M. Georgiev, S. Iyengar, S. Jana, The most dangerous code in the world: validating SSL certificates in non-browser software, in: Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12, 2012, pp. 38–49.
[40] S. Fahl, M. Harbach, T. Muders, Why Eve and Mallory love Android: An analysis of Android SSL (in) security, in: ACM conference on Computer and communications security, 2012, pp. 50–61.
[41] T. Jager, J. Somorovsky, How to break XML encryption, Proceedings of the 18th ACM conference on Computer and communications security - CCS '11 (2011) 413doi:10.1145/2046707.2046756.
[42] J. Rizzo, T. Duong, Practical padding oracle attacks, Proc. of the 4th USENIX conf. on offensive technologies (2010) (2010) 1–9.
[43] T. Duong, J. Rizzo, Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET, EEE Symposium on Security and Privacy (2011) 481–489doi:10.1109/SP.2011.42.
[44] B. Schneier, Cryptographic design vulnerabilities, Computer (September) (1998) 29–33.
[45] P. Gutmann, Lessons Learned in Implementing and Deploying Crypto Software, Usenix Security Symposium.
[46] R. Anderson, Why cryptosystems fail, Proceedings of the 1st ACM conference on Computer and communications security (1993) 215–227.
[47] B. Schneier, Designing Encryption Algorithms for Real People, Proceedings of the 1994 workshop on New security paradigms. (1994) 98–101.
[48] A. Braga, R. Dahab, Mining Cryptography Misuse in Online Forums, in: IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2016, pp. 143–150. doi:10.1109/QRS-C.2016.23.
[49] A. Braga, R. Dahab, A Longitudinal and Retrospective Study on How Developers Misuse Cryptography in Online Communities, in: XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg'17), Brasília, DF, Brazil, 2017.
[50] A. Braga, R. Dahab, N. Antunes, N. Laranjeiro, M. Vieira, Practical Evaluation of Static Code Analysis Tools for Cryptography: Benchmarking Method and Case Study, in: The 28th IEEE International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2017.
[51] P. Rogaway, The Moral Character of Cryptographic Work, Tech. rep., IACR-Cryptology ePrint Archive (2015).
[52] M. Green, M. Smith, Developers are Not the Enemy!: The Need for Usable Security APIs, IEEE Security and Privacy 14 (5) (2016) 40–46. doi:10.1109/MSP.2016.111.
[53] P. Junod, Towards Developer-Proof Cryptography, Tech. rep., EPFL, Summer Research Institute, Lausanne, Switzerland (2016).
[54] C. E. Landwehr, A. R. Bull, J. P. McDermott, W. S. Choi, A Taxonomy of Computer Program Security Flaws, ACM Comput. Surv. 26 (3) (1994) 211–254. doi:10.1145/185403.185412.
[55] T. Aslam, I. Krsul, E. H. Spafford, Use of A Taxonomy of Security Faults.
[56] S. Weber, P. A. Karger, A. Paradkar, A software flaw taxonomy: aiming tools at security, ACM SIGSOFT Software Engineering Notes 30 (4) (2005) 1–7.
[57] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, C. Zhai, Bug characteristics in open source software, Empirical Software Engineering 19 (6) (2014) 1665–1705. doi:10.1007/s10664-013-9258-8.
[58] Z. Wan, D. Lo, X. Xia, L. Cai, Bug characteristics in blockchain systems: a large-scale empirical study, in: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 413–424. doi:10.1109/MSR.2017.59.
[59] Cryptography Coding Standard. URL cryptocoding.net/index.php
[60] C. Koç, About Cryptographic Engineering, 2009.
[61] D. Hankerson, S. Vanstone, A. Menezes, Guide to elliptic curve cryptography, 2004.
[62] J. Katz, Y. Lindell, Introduction to Modern Cryptography.
[63] C. Paar, J. Pelzl, Understanding cryptography: a textbook for students and practitioners, Springer Science & Business Media, 2009.
[64] N. Ferguson, B. Schneier, T. Kohno, Cryptography Engineering: Design Principles and Practical Applications, Wiley, 2011.
[65] J. B. Knudsen, Java Cryptography, Oreilly, 1998.
[66] P. Chandra, M. Messier, J. Viega, Network security with OpenSSL, O'Reily, June.

[67] D. Hook, Beginning cryptography with Java, John Wiley & Sons, 2005.
[68] T. S. Denis, Cryptography for Developers, Syngress Publishing, 2006.
[69] OWASP, OWASP Top Ten Project (2013).
URL https://www.owasp.org/index.php/Top_10
[70] CYBSI, Avoiding The Top 10 Software Security Design Flaws (2014).
URL http://cybersecurity.ieee.org/
[71] Safecode, Fundamental Practices for Secure Software Development (2011).
URL http://www.safecode.org/wp-content/uploads/2014/09/SAFECode_Dev_Practices0211.pdf
[72] SANS/CWE, TOP 25 Most Dangerous Software Errors.
URL www.sans.org/top25-software-errors
[73] OWASP, Cryptographic Storage Cheat Sheet.
URL www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet
[74] A. Chatzikonstantinou, C. Ntantogian, C. Xenakis, G. Karopoulos, Evaluation of Cryptography Usage in Android Applications, 9th EAI International Conference on Bio-inspired Information and Communications Technologies.
[75] A. Braga, R. Dahab, Introdução à Criptografia para Programadores: Evitando Maus Usos da Criptografia em Sistemas de Software, in: Caderno de minicursos do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2015, 2015, pp. 1–50.
[76] S. Das, V. Gopal, K. King, A. Venkatraman, IV= 0 Security Cryptographic Misuse of Libraries, Tech. rep. (2014).
[77] E. S. Alashwali, Cryptographic vulnerabilities in real-life web servers, in: Third International Conference on Communications and Information Technology (ICCIT), Ieee, 2013, pp. 6–11. doi:10.1109/ICCITechnology.2013.6579513.
[78] J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, E. Wustrow, Elliptic curve cryptography in practice, in: Financial Cryptography and Data Security, Springer, 2014, pp. 157–175.
[79] V. G. Mart, L. Hern, Implementing ECC with Java Standard Edition 7, International Journal of Computer Science and Artificial Intelligence 3 (4) (2013) 134–142. doi:10.5963/IJCSAI0304002.
[80] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, Others, Imperfect forward secrecy: How Diffie-Hellman fails in practice, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 5–17.
[81] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, S. Chenjie, Analysis on Password Protection in Android Applications, in: P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on, 2014, pp. 504–507. doi:10.1109/3PGCIC.2014.102.
[82] Oracle, Java Cryptography Architecture (JCA) Reference Guide.
URL docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html
[83] S. Fahl, M. Harbach, H. Perl, Rethinking SSL development in an appified world, Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13 (2013) 49–60.
[84] Oracle Java Cryptography.
URL https://community.oracle.com/community/java/java_security/cryptography
[85] Google Android Developers.
URL https://groups.google.com/forum/#!forum/android-developers
[86] OWASP, List of Source Code Analysis Tools.
URL https://www.owasp.org/index.php/Source_Code_Analysis_Tools
[87] NIST, Software Assurance Metrics And Tool Evaluation (SAMATE).
URL https://samate.nist.gov
[88] UMD, FindBugs.
URL http://findbugs.sourceforge.net
[89] P. Arteau, FindSecBugs.
URL https://find-sec-bugs.github.io
[90] NCCGroup, VisualCodeGrepper.
URL https://github.com/nccgroup/VCG
[91] RigsIT, Xanitizer.
URL https://www.rigs-it.net
[92] SonarSource, SonarQube.
URL https://www.sonarqube.org
[93] M. Scovetta, Yasca.
URL http://yasca.org
[94] L. Sampaio, A. Garcia, Exploring context-sensitive data flow analysis for early vulnerability detection, Journal of Systems and Software 113 (2016) 337 – 361.
[95] A. Delaitre, B. Stivalet, E. Fong, V. Okun, Evaluating bug finders - Test and measurement of static code analyzers, Proc. of 1st International Workshop on Complex Faults and Failures in Large Software Systems, COUFLESS'15 (2015) 14–20doi:10.1109/COUFLESS.2015.10.
[96] OWASP, OWASP Benchmark Project.
URL https://www.owasp.org/index.php/OWASP_Benchmark_Project
[97] G. Díaz, J. R. Bermejo, Static analysis of source code security: Assessment of tools against SAMATE tests, Information and Software Technology 55 (8) (2013) 1462–1476. doi:10.1016/j.infsof.2013.02.005.
[98] K. Goseva-Popstojanova, A. Perhinschi, On the capability of static code analysis to detect security vulnerabilities, Information and Software Technology 68 (2015) 18–33. doi:10.1016/j.infsof.2015.08.002.
[99] A. Braga, R. Dahab, Towards a Methodology for the Development of Secure Cryptographic Software, in: The 2nd International Conference on Software Security and Assurance (ICSSA 2016), 2016.
[100] A. Braga, C. Rubira, R. Dahab, Tropyc: A pattern language for cryptographic object-oriented software, Chapter 16 in Pattern Languages of Program Design 4 (N. Harrison, B. Foote, and, in: Also in Procs. of PLoP, 1999.
[101] A. Braga, D. Schwab, The Use of Acceptance Test-Driven Development to Improve Reliability in the Construction of Cryptographic Software, in: The 9th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2015)., 2015.
[102] A. Braga, E. Morais, Implementation Issues in the Construction of Standard and Non-Standard Cryptography on Android Devices, in: The 8th International Conference on Emerging Security Information, Sys-

tems and Technologies, 2014, pp. 144–150.
[103] A. Braga, E. Nascimento, Portability evaluation of cryptographic libraries on android smartphones, Springer, 2012, pp. 459–469.
[104] S. Clark, T. Goodspeed, P. Metzger, Z. Wasserman, K. Xu, M. Blaze, Why (Special Agent) Johnny (Still) Can'T Encrypt: A Security Analysis of the APCO Project 25 Two-way Radio System, in: Proceedings of the 20th USENIX Conference on Security, SEC'11, USENIX Association, Berkeley, CA, USA, 2011, p. 4.
[105] S. Fahl, M. Harbach, T. Muders, Helping Johnny 2.0 to encrypt his Facebook conversations, in: Proceedings of the Eighth . . . , 2012.
[106] S. Garfinkel, R. Miller, Johnny 2: a user test of key continuity management with S/MIME and Outlook Express, in: Proceedings of the 2005 symposium on Usable . . . , 2005, pp. 13–24.
[107] K. Ermoshina, H. Halpin, F. Musiani, Can Johnny Build a Protocol? Co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols.
[108] M. Sweikata, G. Watson, C. Frank, C. Christensen, Y. Hu, The usability of end user cryptographic products, in: 2009 Information Security Curriculum Development Conference on - InfoSecCD '09, ACM Press, New York, New York, USA, 2009, pp. 55–59. doi:10.1145/1940976.1940988.
[109] S. Arzt, S. Nadi, K. Ali, E. Bodden, S. Erdweg, M. Mezini, Towards Secure Integration of Cryptographic Software, in: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), Onward! 2015, ACM, New York, NY, USA, 2015, pp. 1–13. doi:10.1145/2814228.2814229.
[110] Y. Li, Y. Zhang, J. Li, D. Gu, iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications, in: 8th International Conference on Network and System Security, Springer International Publishing, Xi'an, China, 2014, pp. 349–362. doi:10.1007/978-3-319-11698-3_27.
[111] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, Z. Zhang, Vetting SSL usage in applications with SSLint, in: 2015 IEEE Symposium on Security and Privacy (SP), IEEE, 2015, pp. 519–534.
[112] A. Barenghi, L. Breveglieri, I. Koren, D. Naccache, Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures, in: Proceedings of the IEEE, Vol. 100, IEEE, 2012, pp. 3056–3076.

Chapter 3

Discussion

This thesis is about how to better assist developers in their coding tasks for blending cryptography into software functionality in mashups, sometimes, hard to anticipate. In order to contribute to accomplish this goal, this thesis promotes a systematic approach to cryptographic software security.

This chapter discusses our findings and contributions in relation to accepted practice and established knowledge for secure software engineering, mainly provided by three textbooks, well known to software practitioners: Sommerville (2015) [97], Pressman and Maxin (2014) [85], and the IEEE Software Engineering Body of Knowledge (SWEBOK, 2013) [19]. Besides being good representatives for the established knowledge in secure software engineering, these textbooks were chosen because they explicitly discuss the development of secure software and complement the analysis we did before [30], now targeting knowledge of experts on secure software engineering.

The chapter is organized as follows. Section 3.1 revisits the general motivation for secure software engineering and relates it to the particular motivation for cryptographic software security. Section 3.2 gives an example of a software design that exposes many opportunities for the misuse of cryptography. Section 3.3 explains how the concept of cryptography misuse evolved from two related concepts of code vulnerability and design flaw. Section 3.4 discusses how cryptographic software security is linked to secure software engineering. Section 3.5 discusses cryptographic software security into phases for development of cryptographic software. Finally, Section 3.6 concludes the case for cryptographic software security.

3.1 The need for systematic methods in cryptographic software security

This section analyzes the rationale and motivation to secure software engineering in general and for a methodology for development of secure cryptographic software [28], in particular.

Sommerville [97] distinguishes between application and infrastructure security. For him, application security is a software engineering problem in which a system is designed to resist attacks, while infrastructure security is a systems management problem where

the infrastructure is configured to resist attacks. Also, he believes that the security of a system is a system property that reflects the system's ability to protect itself from accidental or deliberate external attack.

Pressman and Maxin [85] have a process view aimed at collecting evidence for quality assurance. They say that software security is an essential part of software quality assurance for any networked software application. For them, security is the concern of every software engineer and every stakeholder seeking to protect user privacy and intellectual property associated with a computer system. They say that the primary working products are a security specification and a documented security case. To develop these working products, threat models and plans for security risk assessments and risk mitigation are created. The evidence resulting from security reviews, inspections, and test results are presented as a security case that allows system stakeholders to assess their degree of trust in a system.

SWEBOK[19] states that, in addition to the usual correctness and reliability, software developers must also pay attention to the security of the software they develop. Secure software development builds security in software by following a set of established and/or recommended rules and practices in software development. SWEBOK also states that secure software maintenance complements secure software development by ensuring that no security problems are introduced during software maintenance.

In addition, security must also be taken into consideration when performing software maintenance as security faults and loopholes can be and often are introduced during maintenance. Interestingly, in accordance with SWEBOK[19], the Software Extension of the PMBOK Guide [83] provides a process-oriented treatment to software security for software development projects.

The SWEBOK [19] accepts the view that it is much better to design security into software than to patch it after software is developed. To design security into software, one must take into consideration every stage of the software development life cycle [19]. In particular, secure software development involves software requirement security, software design security, software construction security, and software testing security, as well as concerns on risk management and security assurance [19].

Historically, experts in software security systematically avoid entering cryptography issues during development, minimizing them to high-level, theoretic knowledge and occasional advice from experts. For instance, the Open Web Application Security Project (OWASP) [81] recognizes a current focus on high-level guidelines for developers and architects who implement cryptographic solutions.

Cryptologists have a tendency to value theory and neglect practical issues associated to cryptography misuse. For many experts, cryptography issues reside only below APIs, inside libraries, concerned only with the security of algorithm implementation. For instance, in a survey on advanced tools for coding and verification of cryptographic software [25], we found that only one quarter (around of 25%) of all surveyed tools can be applied above crypto APIs, by ordinary developers. The other 75% being applicable only below APIs, for implementation of cryptographic algorithms.

This behavior of cryptologists may have influenced practitioners to react negatively to cryptography issues, sometimes adopting high-level risk analysis as the only assurance

control available and limiting activities to vulnerability management during operations and software patching after system compromise.

This state of affairs also contributed to perpetuate misunderstandings about the correct use of cryptography, resulting in the exploitation of simple vulnerabilities with catastrophic consequences (see [41, 67]), as well as leading to frequent misuse of cryptography [46, 94, 68], improper certificate validation [54, 49], and inappropriate error handling when orchestrating crypto services [61, 86, 45].

Many studies have shown that vulnerabilities in crypto software are mainly caused by software defects and poorly managed parameters [89, 56, 12, 90]. Recent studies [46, 54, 49, 94, 68], including ours [27, 29], showed the occurrence of known crypto misuses in modern software platforms. Also, current tools are unable to properly cope with security issues in programming crypto software [31]. Furthermore, most advice about cryptography in textbooks for software security is obsolete by the standards of today [30].

Recently, cryptography researchers started to analyze the reasons behind this situation. For Ross Anderson [13], there is a social divide between the two communities of cryptology and computer security, because security experts do not always understand cryptographic tools, and cryptologists do not always understand real-world problems [13]. Rogaway [87] claims that cryptologists should acquire a system-level view and attend to what surrounds their field. Also, Green and Smith [55] argue that modern security practice has created an adversarial relationship between security software designers and developers. For them [55], security experts must focus, instead, on creating developer-friendly approaches to strengthen system security (in opposition to a developer-proof attitude [62]). Rogaway [87] also suggests an API-centric approach. For him [87], API misunderstandings and (semantic) gaps between cryptography theory and API design are common causes of problems in cryptographic software.

From the viewpoint of cryptographic software security, without an ordered way (for example, as the one provided by our methodology) to mitigate cryptography misuse in coding, design and architecture, developers are likely to overlook the hardest cryptography issues, underestimating their complexity, and minimizing them to high-level, theoretic knowledge and advice from experts. Therefore, there is a concrete need for methods for building secure cryptographic software, in such a way that cryptographic security could be easily blended into applications' functionality.

It is important to observe that our methodology for development of secure cryptographic software is not a step-by-step on how to apply best practices for specifying, coding, and testing cryptographic software. That would be a methodology of the past, when designs were more intellectually manageable, and the potential interactions among components could be completely planned, understood, anticipated, and guarded against. In that old-school scenario, less incomplete tests were possible, due to a reduced number of possible uses for cryptographic infrastructures in software.

We found [30] that the main approaches for development of secure software fall in this old-school style of methods and were invented when cryptography was not widely available, and limited to few (simple) use cases. Therefore, these approaches are unable to cope with the increasing complexity of modern cryptographic software. Today, many incidents involving cryptography misuse are related to complex interactions involving the

complete software system and the cryptographic subsystem. Thus, traditional development methods cannot describe them adequately.

According to PMI's Agile Practice Guide [84], modern development methods are adaptive to changing requirements and foster continuous learning by developers, which frequently requires experimentation and feedback. Experimentation is important for developers to update their knowledge and better handle changing requirements or evaluating several options for the best solution to an unusual use case. Developers also use feedback to update their knowledge as crypto systems evolve.

In cryptographic software security, the way for developers to determine that their knowledge has to be updated is through experimentation, which requires appropriate techniques and tools for feedback. That is, in order to learn where the boundaries of safe usage are, occasionally developers have to try unsafe designs. Thus, a methodology for development of secure cryptographic software has to provide appropriate feedback to developers in order to correct the course of design, avoiding likely crypto misuses, instead of only prescribing closed solutions to known use cases.

3.2 A running example for designing cryptographic software

This section gives an example of what to watch for in the architecture design of cryptographic software in order to avoid not only coding vulnerabilities, but also design flaws and insecure architectural choices regarding cryptography misuse. The example is non-exhaustive and favors readability.

The manual composition of encryption and MAC (Message Authentication Code) is used to illustrate the iterative development of a **secure enough** composition of encryption and MAC. For the purposes of this example, a MAC function is a keyed-hash function used to provide authentication of content to message exchanges between two communicating parties. Detailed explanations about Message Authentication Codes can be found in specialized books for cryptography [72, 63, 82].

In modern cryptographic software, features for management, distribution, and storage of keys can be blended into application functionality and are influenced by usability requirements and system architecture, being transparent to final users.

Two applications exemplify actual use cases for the manual composition of encryption and MAC. First, a mobile app for authenticated encryption of Short Message Service (SMS) [37], where the combination of encryption and MACs had great potential for design flaws, the transport of shared keys was masked as invitations to be a contact, and key update was blended into user notifications and app updates. Second, a mobile app for encrypted file system with secure deletion of files [24], where secure deletion was obtained by purging crypto keys from storage, and copies of a file could not be deterministically encrypted to be equal. In this single-user file system, the integrity of encrypted files was granted by computing MACs.

In this running example, knowing that a cryptographic software is about to be built, developers may start with the architectural choices regarding the cryptographic library

and randomness sources. These choices are related to the system's structure and are affected by cryptography requirements. These architectural choices can be summarized by the following (non-exhaustive) list of best practices:

- Provide secure (Pseudo-)Random Number Generators (PRNG) as infrastructure.
- Use a single implementation (library) for cryptography, if possible.
- Choose a cryptographic library of good reputation.
- Be prepared to switch implementations if crypto library gets compromised.
- Design and build key distribution/agreement features.
- Design and build key management features.
- Design for secure key storage.
- Design for key life cycles to never use expired keys.

For the scenario of secure communication with SMS, a key agreement protocol can provide forward secrecy and still protect past encrypted messages if the current key is compromised. However, SMS is a best effort message service, without delivery guarantees, and is not reliable for supporting an online key agreement protocol. Thus, an offline key distribution is a better design from the application point of view, but it loses the forward secrecy property.

Evidence for insecure architectural choices can be found in code. For instance, a hard-coded key means that features for management, storage, and distribution of keys have been neglected in system design. Also, many copies in source code of a custom implementation for a cryptographic algorithm mean that homemade cryptography is in use and there is no single point of access for reliable cryptography.

Proceeding to design decisions, developers choose how to structure program elements to construct the composition of encryption and MAC. It's well known by cryptography experts [65] that the only secure way to manually combine encryption and MAC is to compute the authentication tag from the cipher text, using a technique named (Encrypt-then-Mac (EtM)). Other composition techniques are insecure. For instance, the (Encrypt-and-Mac (EaM)) technique computes the tag from the clear text, resulting in a deterministic tag. Also, the (Mac-then-Encrypt (MtE)) technique encrypts the tag along with the clear text, but still results in a deterministic tag. Also, both techniques may expose side channels based on the verification of that tag.

Also, cryptologists strongly recommend the use of single **Authenticated Encryption (AE)** functions, instead of manual composition of encryption and MAC, because AE binds the decryption to the correct verification of the tag. The clear text is unavailable otherwise.

Nevertheless, manual composition of encryption and MAC is still possible because those routines for encryption and MAC are readily available to programmers in many cryptographic libraries. This happens not only because crypto libraries have to stay compatible with past software, but also because it is not feasible to designers of generic

crypto libraries to predict all possible uses developers will make of these libraries in modern cryptographic software.

Proceeding with the example, supposing that developers choose the right technique to build a manual combination of encryption and MAC, (Encrypt-then-Mac (EtM)), there are other design decisions to be made by developers. These design decisions can be summarized by the following (non-exhaustive) list:

- Use different keys for encryption and MAC.
- Correctly select cryptographic primitives.
  - Select secure hash functions.
  - Select secure encryption algorithms (e.g., block ciphers, stream ciphers, etc.).
- Choose good randomness sources to generate nonces, IVs, and keys.
- Use an IV management technique adequate to the operation mode.
- Do not reuse IVs with the same key.
- Include the IV in MAC computation.
- Choose long enough keys with balanced sizes for encryption and MAC.
- Use a secure storage to keep keys safe.
- Avoid timing channels when comparing two MACs.
- Avoid padding oracles when exposing error messages.

Many design flaws can also be found in code. For instance, all design flaws related to insecure selection of cryptographic primitives, insecure key sizes, insecure PRNGs can be found by code review. Insecure block ciphers, misconfigured stream ciphers, broken hash functions, timing channels, padding oracles, hard-coded IVs, and small keys are straightforward on source code.

Finally, proceeding to coding tasks, there are still several concerns to watch before the solution can be considered secure enough. The useful practice for coding cryptographic software for this example can be summarized as follows:

- Correctly select cryptographic schemes from the available infrastructure.
  - Use a MAC built with a secure hash function (e.g., HMAC).
  - Use a secure block cipher for symmetric encryption.
  - Use a secure operation mode for the block cipher.
  - Use key derivation functions (KDF) to generate keys.
- Choose a padding technique for the encryption scheme.
- Choose seeds from selected sources for PRNGs.

- When recovering keys, keep their exposition in code or memory to a minimum.
- Do not decrypt before successfully verifying the authentication tag.

All these code-level decisions related to cryptography programming have counterparts in simple cryptography misuse and can be found by code reviews supported by static analysis tools. However, developers are not trained to look for complex cryptography misuse in software. Thus, design flaws and insecure architectures are likely to pass unnoticed by developers without expert help from cryptologists. In general, developers are exposed only to (good) use cases for cryptography, because, until now, there was no comprehensive knowledge base of cryptography misuse from a developer's point of view. We aim at contributing to fulfill this gap.

3.3 Code vulnerability, design flaw, and cryptography misuse

In this section, we explain how the concept of cryptography misuse is related to the concepts of code vulnerability and design flaw. Information security books (e.g., [13, 57]), as well as software engineering books (e.g., [97, 85]) usually conceptualize vulnerability as a weakness. We do not feel comfortable with the use of this definition for software security because, in the common language used by non experts in security, vulnerability and weakness are synonymous.

In order to address the needs of software developers, adapting the concept to our purposes, we adopt context-aware definitions provided by software security experts. For instance, Merkow and Haghavan [73] say that a vulnerability is a defect in the implementation that opens a pathway for an attacker (with the right set of skills) to exploit the defect and cause the software to behave in ways the developer never anticipated.

Allen et al. [11] adopt the following definitions. A vulnerability is a software defect that an attacker can exploit. For them, defects typically fall into one of two categories: bugs and flaws. A bug is a problem introduced during software implementation (coding tasks). Most bugs can be easily discovered (with tool support) and corrected, while flaws are more subtle, typically originating in the design and being instantiated in the code. Examples of flaws include compartmentalization problems in design, error-handling problems, broken access control, and some kinds of cryptography misuse.

Allen et al. [11] and Sommerville [97] also state that a failure — an externally observable event — occurs when a system does not deliver its expected service (as specified or desired), while an error is an internal state that may lead to failure if the system does not handle the situation correctly; a fault is the cause of an error. For them, the functional error can be leveraged into a security failure, and a number of errors can be demonstrated to lead to exploitable failures.

McGraw [71] explains that flaws are often much more subtle than code-level vulnerabilities, such as, for instance, a simple off-by-one error in an array reference. A flaw is certainly instantiated in software code, but this code-level issue is in fact an expression of a flaw present at the design level. For instance, a hard-coded key found in source code is a

consequence of missing features for key management that were not designed. McGraw [71] also warns that automated technologies to detect design-level flaws do not yet exist *(by that time)*, though manual processes (inspections) can identify flaws.

McGraw [71] distinguishes flaws from bugs based on how much program code must be considered to understand the vulnerability, how much detail regarding the execution environment must be known to understand the vulnerability, and whether a design-level description is best for determining whether or not a given vulnerability is present. He also argues that mid-range vulnerabilities involve interactions among more than one location in code, and design-level vulnerabilities carry this trend further. He warns that ascertaining whether or not a program has design-level vulnerabilities requires great expertise, because these flaws are hard to find and particularly hard to automate.

As we stated before [30] and complemented by the above arguments, a **cryptography misuse** (crypto misuse, for short) is a programming bad practice frequently found in cryptographic software, leading to vulnerabilities, and introduced by developers during coding tasks associated to use cases enabled by cryptography. We do not simply name these misuses vulnerabilities because, in many cases, they are design flaws and insecure architectural choices. Crypto misuse is not related to implementation of cryptographic algorithms. Instead, crypto misuses emerge when ordinary developers use cryptographic infrastructures in their daily coding activities during the development of cryptographic software.

The concept of cryptography misuse adopted in this thesis is in accordance with several related works in the recent literature. For instance, Lazar et al. [68] found that only 17% of cryptography vulnerabilities are inside software libraries, the other 83% are misuse of libraries, while Egele et al. [46] and Chatzikonstantinou et al. [38], in two different studies, found that about 88% of mobile apps (from Google's market place) showed some cryptography misuse. Also, Gajrani et al. [52] found that 90% of apps from diverse app stores are exploitable because of cryptographic vulnerabilities. Our own results [27] confirmed these high values for cryptography misuse. We found that cryptography misuse appears in regular discussions about cryptographic programming in online communities in 90% of posts for Java and 71% of posts for Android.

Also, related works found instances of cryptography misuse quite similar to the ones we found in our investigation. For instance, Egele et al. [46] and Shuai et al. [94] found that deterministic encryption is the most common misuse when a block cipher (e.g., AES or 3DES) uses the Electronic Code Book (ECB) mode. Asymmetric deterministic encryption with non-randomized RSA is also a misuse [56]. Hardcoded or repeated Initialization Vectors (IV) and hard-coded seeds for Pseudo-random Number Generators (PRNGs) are also frequent [46]. Other misuses come from exchanging operation modes without considering IV needs [27].

Additionally, Georgiev et al. [54] and Fahl et al. [49] showed that libraries for SSL/TLS allow programmers to ignore parts of certificate validation, adding vulnerabilities exploitable by man-in-the-middle attacks. Recent studies showed misuses related to weak or misplaced parameters for RSA [8], key agreement misconfiguration (e.g., DH and ECDH) [7], and Elliptic Curve Cryptography (ECC) [18, 70] as well.

Furthermore, Nadi et al. [75] observed that developers usually implement simple use

cases (e.g. user authentication, storage of login data, secure connections, and data encryption), but face difficulties when using low-level Java APIs. Also, Shuai et al. [93] discovered that password protection in Android is greatly affected by cryptography misuse.

In our own investigation [27], we found that the most widespread misuse is weak cryptography, affecting several crypto use cases, and the most troublesome use case is encrypting data at rest, which is affected by several misuses.

We also found that the different kinds of cryptography misuse are not equally difficult to avoid. Also, from the developer's point of view, some misuses are easier to correct than others. For instance, low-complexity misuse is related to code-level vulnerabilities, and could be easily found by early detection techniques, simple code reviews, and skilled developers (supported by tools). We showed [27, 29, 31] that this kind of misuse includes Weak Cryptography (WC), Coding and Implementation Bugs (CIB), and Bad Randomness (BR) manipulation. We consider these three kinds of misuse are of low complexity because no deep understanding of program design is required to mitigate them, as well as fixes are likely to be related to single programs or simple code snippets.

Medium-complexity misuse is related to design flaws affecting a few different programs and may be difficult to identify due to feature distribution across programs. We showed [27, 29, 31] that this kind of misuse includes Improper Certificate Validation (ICV) issues, Program Design Flaws (PDF), and Public-Key Crypto (PKC) issues. We consider these three kinds of misuse are of medium complexity because fixing them may require program redesign, possibly affecting a few programs. Avoiding them requires more knowledgeable developers and support from experts.

High-complexity misuse is related to insecure architectural choices, and requires understanding of system architecture to analyze underlying cryptosystems. We showed [27, 29, 31] that this kind of misuse includes Poor Key Management (PKM), IV and Nonce Management (IVM) issues, and Crypto Architecture and Issues (CAI). We consider these three kinds of misuse are of high complexity because fixing them usually require new modules or redesign of modules, and may affect many code bases. These misuses require cryptography experts to perform code and design reviews, or architecture analysis.

3.4 Cryptographic software security and secure software engineering

This section analyzes how secure software engineering links to cryptographic software security. This section is organized as follows. Subsection 3.4.1 relates the concepts of secure software engineering and cryptographic software security. Then, two subsections discuss risk management (Subsection 3.4.3) and security assurance (Subsection 3.4.2) in the context of cryptographic software security.

3.4.1 Secure software engineering and secure cryptographic software

For the purposes of this text, Secure Software Engineering, also named Software Security [71], software engineering security [19], and (software) security engineering [97], is the process of designing, building, and testing software so that it becomes secure [64].

According to Sommerville [97], security engineering is the set of tools, techniques, and methods to support the development and maintenance of systems that can resist malicious attacks intended to damage a computer-based system or its data. For him, the main concern of security engineering is to develop systems that can resist malicious attacks. Also, he considers software security as a sub-field of the broader field of computer security.

According to Pressman and Maxin [85], security is a requisite for system integrity, availability, reliability, and safety, because it provides the mechanisms that enable a system to protect its valuable resources (e.g., information, files, programs, storage, processor capacity) from attacks which take advantage of vulnerabilities that allow unauthorized system access. Pressman and Maxin [85] complement that it is difficult to make software more secure by just responding to bug reports, because security must be designed in from the beginning.

Also, according to the SWEBOK [19], any software is only as secure as its development process goes. So, to ensure the security of software, security must be built into the software engineering process. The SWEBOK [19] identifies the Secure Software Development Life Cycle (SSDLC) as an emerging trend structured as a classical spiral model, that takes a holistic view of security from the perspective of software life cycles, and ensures that security is inherent in software design and development, not an afterthought later in production. Such an SDLC process is claimed to reduce software maintenance costs and increase reliability of software, when concerned to faults related to software security.

In general, software security does not directly address the issues of cryptographic security because cryptography is considered a security feature effectively added to software during coding, with no specific support from development methods [71, 13]. Historically, software security favored penetration testing against known vulnerabilities of standard protocols, enforcement of simple, high-level coding guidelines against misuse of crypto libraries, and compliance to general security policies.

We argue that software security methods can be enhanced to empower developers with proper ways to avoid cryptography misuse when building crypto software. This enhancement can be obtained by adopting the knowledge base of the emerging discipline of cryptographic software security, a sub-field of software security, targeting three complementary aspects: a methodology for development of secure cryptographic software, a classification of cryptography misuse for software security, and a reference layered architecture for cryptographic software.

Over the years, we have observed the frequent adoption of common practices by researchers and practitioners, which gave us the required perspective to generalize a working methodology for development of secure cryptographic software. Our methodology [28], emerged as a response to practical needs observed when we: (i) documented design pat-

terns for cryptography [20]; (ii) built cryptographically secure mobile apps [34, 37, 24, 36]; (iii) applied Acceptance Test Driven Development (ATDD) to cryptographic services [35] (when building [32] and porting [33] a cryptography library to mobile devices); (iv) surveyed tools for secure programming and verification of cryptographic software [25]; (v) trained developers in cryptography programming [26]; (vi) investigated crypto misuses in online communities [27, 29]; and (vii) evaluated static code analysis tools for cryptography [31].

3.4.2 Software assurance for cryptographic software security

For Sommerville [97], security is related to assurance of safety and reliability. For example, statements about reliability and safety of an insecure system are unreliable because these statements depend on the executing system and the developed system being the same. However, intrusion can change the executing system and/or its data. Therefore, the reliability and safety assurance is no longer valid.

Pressman and Maxin [85] prefer the concept of system trustworthiness, in a way that trust is the level of confidence that software components and stakeholders (e.g., developers) can rely on one another. Verification ensures that the security requirements are assessed using objective and quantifiable techniques traceable to the security cases. Evidence used to prove the security case must be acceptable and convincing to all system stakeholders. Most trust metrics are based on historical data derived from past behavior. Complementing the above statements, for McGraw [71], the confidence in software security is relative to the assurance methods in use.

In general, cryptologists [50] understand that cryptographic algorithms are supposed to be perfect mathematical constructs (ideally), but are imperfect in practice due to implementation constraints. Also, in systems with many building blocks, cryptography practitioners have to check whether the combination of imperfect constructions leads to security problems. Due to many combinations, this can be unworkable in practice.

The same cryptologists [50] argue that modular designs for cryptographic algorithms can help assure local correctness. For them [50], cryptographic security is achieved by the local correctness of cryptographic building blocks, that should work correctly no matter what happens in the outside environment, or with the interactions with other building blocks. This viewpoint comes from developers of cryptographic algorithms, which work inside crypto implementations and below crypto APIs.

We believe that this mindset is incomplete and dangerous, because it minimizes (maybe to the point of neglecting) the unexpected and unpredictable interactions among components and how they influence the work of other components in the overall system. This viewpoint does not see cryptographic security as an emerging property of cryptographic software and is obsolete in relation to the current understanding of the phenomenon of cryptography misuse in modern cryptographic software.

In modern crypto software, secure cryptographic functions are not enough for system security, because the reliability of single cryptographic functions is not enough to assure the reliability of whole software systems. In this context, code-level issues and design flaws are causes of problems, even when cryptographic functions work properly and behave as

expected, the issues emerge from unexpected interactions among system components and cryptographic functions in flawed designs.

For instance, when developers do not follow best practices and guidelines for secure coding when using crypto libraries, they are being unsafe and producing insecure cryptographic software. However, with generic APIs, which does not restrict developers options, just following API's usage prescriptions is not enough to code safely, because the resulting code can be insecure at the system level. Cases of misconfigured algorithms as well as unsafe compositions of crypto components are in this situation.

Reliability of crypto components is not enough for its safe use. Their correct usage has to be assured by high-level techniques, above component usage. For instance, a secure design can be made insecure when crypto components, even behaving reliably from an API point of view, are in fact leaking information through some sort of side channel, or showing flaws in accessory subsystems for key management, nonce management, and infrastructure management.

One of the byproducts of this thesis is the understanding that cryptographic security is not a component property (such as, for instance, reliability) that can be attributed to a software library disconnected from its surrounding software. Instead, cryptographic security is an emerging property of systems (such as, for instance, safety), which can be only determined in the context of the whole system and its environment. Determining whether a cryptographic software is secure by looking only at its implementations of cryptography is not enough, and seems meaningless without the context of usage for such implementations.

Inversely, cryptographic security is determined by how a cryptographic library interacts with itself in different moments as well as with the environment (surrounding software). The emergence of cryptographic security comes from the notion of hierarchical levels where constrains at higher levels control or allow behavior at lower levels. Thus, cryptographic security depends on the enforcement of constraints on the behavior of components of the system, including constraints on their potential interactions.

In cryptographic software security, the classification of cryptography misuse can provide the reference checklist to be used for assurance control against cryptography misuse. Also, a reference architecture for cryptographic software, as the one we proposed [28], can be used for validating the correct placement of security controls. Also, recommendations to better use static code analysis tools (like ours [31]) can be used when supporting novice developers to detect the early occurrence of crypto misuse, as well as security experts in inspecting code looking for complex misuses.

Cryptographic software security requires a classification of recurring misuses of cryptography in software, over which assurance methods could be built upon. A thorough understanding of the characteristics of cryptography misuse, captured by a working classification, and targeting the viewpoint of developers, is required to design effective tools for preventing, detecting, and mitigating these misuses early in the construction of software systems.

The classification of cryptography misuse for software security is a core component of the assurance framework for cryptographic software security, which is also composed of other three complementary components:

- a methodology for developing secure cryptographic software.
- a reference architecture for cryptographyc software.
- guidelines on the use of static analysis tools, along with expert support, in specific development contexts.

None of the existing works regarding taxonomies of security issues [66, 15, 106, 101, 105] organized cryptography misuse by characteristics that were useful for our needs. None of these works focused on cryptography misuse from a software security point of view. Therefore, a domain-specific classification was needed to support our work. However, we were not interested in advice for secure implementation of cryptographic algorithms. Instead, we looked for programming techniques for building secure cryptographic software.

The classification of cryptography misuse for software security captures how software developers actually misuse cryptography. Therefore, it is inclined to a software security viewpoint. The current version of the classification of cryptography misuse for software security [30] assembles cryptography misuse in software collected from various sources: literature on software security (e.g., [104, 58, 39, 58, 59, 92]), studies on cryptography misuse (e.g, [68, 38, 46, 94, 26, 54, 49]), discovered misuses (e.g., [8, 18, 70, 7]), and industry initiatives for software security (e.g., [88, 79, 43]).

The classification has been proposed in one of our firsts contributions [27] and refined to fit new findings and observations in other contributions [31, 29]. Its current state of maturity was detailed in a recent contribution [30]. The classification has nine main categories that were synthesized from recommendations covered by literature and are supposed to capture the state of practice: Weak Cryptography (WC), Bad Randomness (BR), Coding and Implementation Bugs (CIB), Program Design Flaws (PDF), Improper Certificate Validation (ICV), Public-Key Cryptography (PKC) issues, IV/Nonce Management (IVM) issues, Poor Key Management (PKM), and Cryptography Architecture Issues (CAI).

3.4.3 Risk management for cryptographic software security

Risk management for security is not standard practice in software engineering. However, it is recognized by SWEBOK [19] that particular attention should be paid to the management of risks related to software quality requirements such as safety or security. SWEBOK [19] relates concepts of risk and uncertainty, because risk is often the result of uncertainty, which results from lack of information necessary to evaluate risk.

According to SWEBOK [19], risk management entails identification of risk factors and analysis of the probability and potential impact of each risk factor, prioritization of risk factors, and development of risk mitigation strategies to reduce the probability and minimize the negative impact, when a risk factor becomes reality. For the SWEBOK [19], risk assessment methods (for example, expert judgment, historical data, decision trees, and process simulations) can sometimes be used in order to identify and evaluate risk factors.

Pressman and Maxin [85] seem not to be particularly interested in risk management because (for them) it is not directly related to assurance. They simply state the commonly

accepted steps for security risk analysis: Identify assets, create architecture overview, application decomposition, identify threats, document threats, and rate threats. On the other hand, Sommerville [97] gives a corporate view to risk management when he says that security risk analysis is a business rather than a technical process. He argues that, in organizations, security is a business issue because security is expensive and it is important that security decisions are made in a cost-effective way. He argues that there is no point in spending more than the value of an asset to keep that asset secure, and organizations use a risk-based approach to support security decision making and should have a defined security policy based on security risk analysis.

For Sommerville [97], security risk management is concerned with assessing possible losses from attacks and deriving security requirements to minimize losses. That way, risk assessment and management is concerned with assessing the possible losses that might result from attacks on the system and balancing the losses against the costs of security procedures that may reduce these losses.

Sommerville [97] explains that risk management happens in three different moments during system development, with different levels of detail: preliminary or business level, life cycle or design, and operational. In a preliminary risk assessment, he says, the aim is to identify generic risks to the system and to decide if an adequate level of security can be achieved at a reasonable cost. This risk assessment helps identify security requirements.

Sommerville [97] details that a life cycle (design) risk assessment, takes place during the system development life cycle and is informed by the technical system design and implementation decisions. The results of this assessment may lead to changes to the security requirements and the addition of new requirements. Also, knowledge on vulnerabilities is used to inform decision makers about the system functionality and how it is to be implemented, tested, and deployed. Finally, Sommerville [97] explains that operational risk assessment focuses on the use of the system and the possible risks that can arise from human behavior. For him, a security risk assessment encompasses the following steps: Asset identification, asset value assessment, exposure assessment, threat identification, attack assessment, control identification, feasibility assessment, and security requirements definition.

In cryptographic software security, there is no particular interest in applying fine-grained risk analysis into activities for mitigation of cryptography misuse, because all potential misuses have to be avoided anyway. This attitude leaves risk analysis at the business level. This way, cryptography misuse, as a broad threat with high impact, will always receive a high score for risk.

In this case, risk is gradually reduced over time by the evolution of understanding about how likely crypto misuses can be avoided in specific use cases and coding tasks. Two of our contributions studied how likely crypto misuse is in use cases and coding tasks for cryptography [27] and how likely static code analysis tools can detect crypto misuse for the same use cases and coding tasks [30].

3.5 A systematic approach to cryptography in software security

This section relates phases of our methodology for development of secure cryptographic software [28] to the common phases for secure software engineering [19]: software requirements security (Subsection 3.5.1), software design security (Subsection 3.5.2), software construction security (Subsection 3.5.3), and software testing security (Subsection 3.5.4).

3.5.1 Software requirements for cryptographic software security

In a simple way, Sommerville [97] says that, to specify security requirements, one should identify the assets that are to be protected, as well as define how security techniques and technology should be used to protect these assets.

Pressman and Maxin [85] explain that the elicitation of security requirements consists in determine how users interact with system resources, create abuser stories that describe system threats, perform user threat modeling and risk analysis to determine the system security policies as part of the non-functional requirements, and identify solutions to system security shortcomings. For them [85], threat analysis is the process of determining the conditions or threats that may damage system resources or make them inaccessible to unauthorized access.

Converging, SWEBOK [19] establishes that software requirements security is the clarification and specification of security policies and objectives into software requirements. Factors to consider in this phase include security software requirements (specific functions that are required for the sake of security) and threats/risks (the possible ways that the security of software is threatened) [19].

In cryptographic software security, the application domain defines the cryptography requirements [28], which, in general, are satisfied by (but not limited to) traditional use cases associated to cryptographic services [27]. These use cases are implemented by coding tasks specific to cryptographic software [27]. In general, cryptography misuse is potentially introduced by developers into use cases during coding tasks [27] as a side effect of technology misunderstandings, semantic gaps in instances of cryptography concepts in APIs and security tools, as well as bad coding practices applied by ordinary developers.

General security requirements for cryptographic modules can be obtained from standards, such as the FIPS-140 [78], which includes not only high-level functional requirements for cryptographic modules in software only or hardware and software, but also non-functional, security requirements that will be satisfied by a cryptographic module utilized within a security system protecting sensitive information. The security requirements cover areas related to the secure design and implementation of a cryptographic module, which is understood as an implementation of security functions, including but not limited to cryptographic algorithms and key generation, and is contained within a defined perimeter.

It is interesting to notice that standards like FIPS-140 [78] do not cover the requirements for cryptographic software that uses a cryptographic module as a building block for application's cryptographic-based security. On the other hand, other documents, such

as ENISA's reports on cryptographic algorithms, key sizes, and protocols [5, 96], provide guidance to developers on how to choose appropriate cryptographic functions and their security parameters when using a trusted cryptographic module. However, these reports are still far from being easily understood by ordinary developers, because they were written with the mindset of a cryptography expert. These reports should be distilled before used to generate cryptographic requirements.

On the other hand, industry-led guidance, such as the guidelines given by OWASP [79], CYBSI [43], and Safecode [88], are targeted to ordinary developers, but do not cover the subject with appropriate depth to avoid crypto misuse.

Our methodology for developing secure cryptographic software can be used to watch for potential crypto misuses related to architectural choices (e.g., crypto agility issues, PKI issues, multiple access points, and randomness sources) and key management (e.g., PBE passwords, expired keys, key distribution, and CA issues), which can be identified and avoided in this phase. Also, a classification of cryptography misuse can be used as a policy to be enforced as well as a checklist of items to be avoided, helping to mitigate cryptography misuse as early as in requirements elicitation.

3.5.2 Software design for cryptographic software security

Sommerville [97] states that the two key issues when designing a secure architecture are organizing the system structure to protect key assets and distributing the system assets to minimize the losses from a successful attack. He believes that it is very difficult to make an insecure system secure after it has been designed or implemented. So, security should be designed into a system considering architectural design and good practices.

Sommerville [97] also sees benefits in the use of design guidelines for security engineering, because guidelines encapsulate good practice in secure systems design, raise awareness of security issues in team members, and provide a basis for a review checklist applied during validation. Finally, he advises that performing a risk assessment while the system is being designed is good practice, because more information is available in design than during requirements gathering, and vulnerabilities that arise from design choices may therefore be identified.

Pressman and Maxin [85] promote security modeling as a way to capture policy objectives, external interface requirements, software security requirements, rules of operation, and description of security architecture; providing guidance during design, coding, and review. Having concerns with security assurance, Pressman and Maxin [85] argue that design should be measured with security metrics, which focus on system dependability, trustworthiness, and survivability. For them, measures for asset value, threat likelihood, and system vulnerability are needed to create these metrics. Also, a security risk analysis is done in design because it is based upon architecture overview and application decomposition.

Promoting a practical viewpoint, SWEBOK [19] states that design for security is concerned with how to prevent unauthorized disclosure, tempering (e.g., creation, change, deletion), or denial of access to information and other resources. SWEBOK [19] is also concerned with how to tolerate security-related attacks or violations by limiting damage,

continuing service, speeding repair and recovery, and failing and recovering securely.

For the SWEBOK [19], software design security deals with the design of software modules that fit together to meet the security objectives specified in the security requirements. Therefore, the design step clarifies the details of security considerations and develops the specific steps for implementation. Here in, factors considered may include frameworks and access modes that set up the overall security monitoring/enforcement strategies, as well as mechanisms for policy enforcement.

In cryptographic software security, some crypto misuses related to architecture (e.g., IV/nonce management issues, API misunderstandings, improper key length, and reused keys), many misuses related to design (e.g., design flaws, certificate validation issues, and PKC issues), and a few misuses related to coding (e.g, proprietary cryptography, custom implementation, and risky/broken cryptography) can be identified and avoided in design phase.

In design phase, attention have to be taken to explicitly design for key management, nonce management, and crypto infrastructure management. Also, modern crypto software has to follow software engineering best practices, including design patterns for cryptographic systems [20] and a layered architecture for cryptographic software [28, 30].

We believe that by designing for management aspects of crypto systems and by adopting a layered architecture, the separation of concerns between simple functions of cryptography and high-level behavior of crypto systems can be made explicit in software designs. This way, constraints on the behavior of low-level components (e.g., crypto APIs) and their interactions will be imposed by high-level components, reducing allowed behavior to what is required by application functionality and thus limiting opportunities for cryptography misuse.

3.5.3 Software construction for cryptographic software security

Showing a strong bias in favor of tools and techniques, SWEBOK [19] is the only one of the three textbooks to explicitly enforce construction for security. For the SWEBOK [19], software construction security concerns the question of how to write actual programming code for specific situations such that security considerations are taken care of. SWEBOK [19] warns that the term "Software Construction Security" could mean different things for different people. It can mean the way a specific function is coded, such that the coding itself is secure, or it can mean the coding of security into software. The book [19] also explains that today software construction security mostly refers to the coding of security into software, which can be achieved by following recommended coding rules.

For the SWEBOK [19], defensive programming is different from secure programming: Defensive programming means to protect a routine from being broken by invalid inputs. Secure coding is called dependable programming by Sommervile[97] and is related to programming practices that support fault avoidance, fault detection, and fault tolerance.

In cryptographic software security, not only the classification of cryptography misuse can be adopted as a coding standard (enforced by static code analysis tools and inspected by experts), but also many crypto misuses related to program design and all misuses related to coding can be identified and avoided in this phase.

We found [27, 29, 31] that mitigation of crypto misuse in coding can be quite straightforward, because coding practices are likely to be simple and related to single programs or simple code snippets, while mitigation of crypto misuse in design may require program redesign and may affect a few programs. Avoiding them requires more knowledgeable developers and support from experts. Also, mitigation of crypto misuse in architecture usually require new modules or redesign of modules, and may affect many code bases. These misuses require cryptography experts to perform code and design reviews, or architecture analysis.

In the construction phase, the development of cryptographic software can obtain most benefit from the adoption of static code analysis tools. Interestingly, roughly half of crypto misuse instances in our test suite could be precisely detected by simple pattern matching. This suggests that the effectiveness of static analysis tools in finding cryptography misuse could be improved just by adding rules to the pattern matching engine already applied by tools.

Also, we noticed that static analysis tools seems to not obtain the full benefit of advanced techniques for vulnerability detection, such as data-flow analysis, to identify cryptography misuse. Only a few tools, of the evaluated tool set, applied data-flow analysis to detect cryptography misuse in quite limited ways, for just a couple of rules. We believe this is due to a knowledge gap between tool builders' current repertoire of actual cryptography misuse and cryptologists knowledge about potential vulnerabilities in crypto systems.

In spite of these limitations, early detection techniques have the potential to contribute to reduce crypto misuse by promptly suggesting fixes to code-level cryptography misuse while code is being written by developers in IDEs. Also, late detection techniques can contribute to the work of reviewers and auditors if, in short term, tools improve recall and precision for detection of cryptography misuse.

3.5.4 Software testing for cryptographic software security

For Sommerville [97], security testing verifies the extent to which the system can protect itself from external attacks. He also notes two problems with security testing. First, security validation is difficult because security requirements state what should not happen in a system, rather than what should. Second, system attackers may have more time to probe for weaknesses than is available for security testing.

Pressman and Maxin [85] enforce correctness checks for assurance, in such a way that software verification activities and security test cases must be traceable to system security cases, and data collected during audits, inspections, and test cases are analyzed and summarized as a security case.

Finally, for SWEBOK [19], security testing is focused on the verification that the software is protected from external attacks. Usually, security testing includes verification against misuse and abuse (negative testing), as well as determines that the targeted software protects its data and maintains security specification as given (by requirements).

For Ferguson, Schneier, and Kohno [50], security is the absence of (unauthorized) functionality used by the unauthorized user (e.g., the attacker). Besides being unable to

prove the absence of defects, tests cannot prove the absence of unauthorized functionality. In spite of that, is still necessary to test software, even not knowing how to do it perfectly or completely for complex software system.

In cryptographic software security, cryptography-related functionality and packages are submitted to functional security tests (supported by security-inspired test cases) and penetration tests, supported by attack scenarios and threats. Likelihood of crypto misuses in use cases can help to identify threats and define attack scenarios. Also, verification includes automated tests for crypto misuses related to coding and program design, as well as security inspections (supported by tools) for crypto misuses related to system design and architecture.

During the development of this thesis, we observed that the following types of dynamic tests for cryptography have been adopted by industry.

- Functional tests as test vectors used to validate implementations of cryptographic algorithms against their specifications, such as test vectors for the Cryptographic Module Validation Program (CMVP) [76], which can be applied when adopting Acceptance Test-Driven Development (ATDD) during the coding of cryptographic algorithms [35].

- Tests against known vulnerabilities of cryptographic algorithms that can compromise the security provided by cryptographic libraries. For instance, the Project Wycheproof [17] is a collection of unit tests that detect known weaknesses or check for expected behaviors of a cryptographic algorithm.

- Automated tests of SSL connections for detection of unencrypted HTTP channels, insufficient TLS protection, and padding oracles, represented by test procedures from OWASP's testing guide[81] as well as prototype tools like MalloDroid [49] and Padding Oracle Exploitation Tool (POET) [86].

These tests sit midway between testing algorithm implementations and testing misbehavior of misconfigured API calls. Semantic tests are application logic's dependent and have to be customized for each cryptographic software and its use cases. In this regard, our classification can help the development of test cases for likely crypto misuse.

Dynamic tests are associated to penetration tests and have an important role in detecting vulnerabilities in deployed cryptographic software, but contribute little to avoid cryptography misuse in source code. Penetration tests for cryptographic software reflect that reactive attitude of postponing the evaluation of cryptographic security until the last security assessment, before releasing software to production or delivery, when crypto misuses related to design or architecture are not only hard to identify, but also expensive to fix.

The reactive approach to security limits further analysis of incidents to those causal events (causes) related to production environments and operation, promoting countermeasures related to operation, and almost always ignoring the fundamental root cause of poorly designed software as a fact of life. The analysis stops at the first event for which an operational or production countermeasure can be adopted.

Also, penetration tests are good for promoting security awareness, but they lack the ability to teach developers how to fix or avoid the vulnerabilities found or demonstrated by an ethical hacker. On the other hand, the concept of cryptography misuse can contribute to educate developers on how to avoid all those design flaws and coding pitfalls frequently related to cryptographic software, increasing awareness in the correct usage of cryptography, without requiring from developers an adversarial mindset.

3.6 The case for cryptographic software security

Cryptography is a social technology used to protect human affairs mediated by this technology. Above every cryptosystem, there are layers of social systems that provide goals, purpose, and constraints to the underlying crypto system. One of those layers is the community of developers working to build and improve the cryptographic software composing and surrounding the crypto system. According to this argument, a cryptosystem is a socio-technical system [69].

In general, there is a tendency to blame the human involved in security vulnerabilities. We observe this parallel also in the development of cryptographic software. Formerly, the operator (final user) used to be the preferred culprit in a cryptography misuse. Then, as soon as cryptography became transparent to users, thanks to better, user-friendly interfaces, the developer was the one to blame. However, after understanding that tools are sadly imperfect and processes barely tackle cryptography misuse, we argue that even developers cannot be solely blamed for the widespread misuse of cryptography in software.

Blaming the developer is easy and comes from the fact that developer errors are the closer events to code-level cryptography misuse. Flawed designs and flawed processes have a not-so-close causal relationship to cryptography misuse, breaking this hindsight bias. In this thesis, we learned that developer's knowledge about cryptography APIs is not only based upon training, but also in the experience of others, with the latter complementing the limitations of the former, for good or for bad.

Developers also use feedback to update their knowledge as cryptosystems evolve. We found that online communities for programming and automated tools for static analysis of code were unable to provide appropriate feedback to developers about cryptography misuse. Experimentation is important for developers to update their knowledge and better handle changing requirements or evaluating several options for the best solution to an unusual use case. Effective communities and tools supporting developers would have to close the experimentation loop, updating developers' knowledge about cryptography misuse and how to avoid it.

Thus, the actual problem resides in providing the appropriate feedback to developers, with smarter tools and supporting communities, allowing experimentation in design to improve developers' ability in avoiding crypto misuse. The missing link, the connection absent in many methods for development of secure software, is the proper feedback from the environment to assist developers in making safe and secure choices about the use of cryptography.

In the cryptographic software of the past, experimentation and feedback were less

important because crypto use cases and coding tasks were simple compared to today's unusual use cases, where cryptography is blended to functionality and transparent to final users at the same time. This blending of cryptography to system functions makes local decisions regarding the security of a cryptography design much more difficult.

It is well accepted by the software engineering community that security has to be treated since the very beginning of software development. However, software security is not self-justified; instead, it is mostly related to regulatory obligations, quality assurance needs, and safety requirements, as well as enforced or restricted by business goals, hopefully without degrading other non-functional requirements, such as performance or usability.

Additionally, cryptographic security must be considered in every stage of the software development life cycle, including software requirement security, software design security, software construction security, and software testing security. This way, a well performed SSDLC can give strong evidence of compliance to security requirements and add justifiable trust to the resulting software of the cryptosystem. Finally, in order to be cost-effective for businesses, cryptographic software security must be supported by automated tools, as well as subjected to continuous feedback during all SDLC phases.

In cryptographic software security, potential misuses of cryptography can be identified early in the development of secure cryptographic software, being mitigated in design, coding, and testing, as well as monitored after the software is released for deployment and put into operation.

The field of cryptographic software security supports the proper use of cryptography in the development of secure software systems by applying a conceptual framework composed of four complementary aspects:

- a methodology for developing secure cryptographic software;
- a classification of cryptography misuse for software security;
- a reference architecture for cryptographic software;
- a set of guidelines on how to use current static analysis tools to detect cryptography misuse.

Chapter 4

Conclusion

We (cryptography experts) have released a Pandora's box and developers have opened it. Today, the use of cryptography is common in software systems and, finally, cryptographic services are widely available to every software developer. However, there are many opportunities for misusing cryptography, as shown by our classification of cryptography misuse, and developers have been very creative in finding them, as evidenced by our findings when investigating online communities for programming. On the other hand, security experts and tool builders have been ineffective in promoting the proper use of cryptography in software systems, as we saw when benchmarking static analysis tools.

We believe that, ideally, developers do not have to learn all the tricky details in order to correctly use cryptography. However, in reality, training developers in using crypto APIs is a partial solution to the problem of cryptography misuse, since it assumes that APIs are well designed from a usability point of view, behave as expected, decoupled from cryptographic algorithms' internals, and tools can precisely detect all deviations from guidelines. As we found in our investigation, this is not the case.

It is our duty, as a community of cryptography experts, to assist developers in the correct use of cryptographic technology. The recognition of cryptographic software security as a serious field of study and a legitimate topic of research can contribute to accomplish this goal.

In this investigation, we were able to clearly see the evolution of understanding regarding cryptography misuse in software systems. At the beginning, before cryptography misuse was even considered a wide-spread, real-world issue by experts in the field, researchers noticed that the misuse of cryptography was associated to serious security breaches.

A lack of reference materials and the advent of attacks exploiting cryptography misuse in actual deployed software (for instance [41, 67, 4, 42, 10, 3, 74, 103, 2, 9]) motivated the study of code vulnerabilities by an emergent community of researchers and practitioners. A great number of vulnerabilities were analyzed and categorized. Knowledge on those vulnerabilities were made available to the community as libraries of programming bad practices, catalogs of simple countermeasures, and high-level design principles.

Soon, researchers and practitioners realized that there was no ultimate security control to avoid cryptography misuse and then redirected efforts to pragmatic solutions and exploratory research. For instance, giving developers the opportunity to educate themselves on the correct use of cryptography is not enough to avoid cryptography misuse. It

is also necessary to provide appropriate feedback to software developers when applying cryptography in specific development environments.

Vulnerability research in applied cryptography is a never-ending effort and now cryptography experts also face the challenge of giving feedback to developers in effective ways and appropriate tool support. We believe that the next step to be taken by this community is to approach the problem of cryptography misuse in software in a systematic way. This thesis contribute to better understand this systematization effort.

We believe that cryptography misuse seems to be pervasive in software and frequent in communities for software development, because cryptographic software security was not approached in a systematic way by software development methods, techniques, and tools. Our findings suggest that those people involved in software development overlook cryptography misuse because, in general, it is hard to identify with current detection tools, except for simple cases,

Based on our findings, we believe that cryptographic software security needs a new generation of supporting technology composed not only by verification tools, APIs, and development frameworks, but also by appropriate development methods able to foster effective feedback to developers regarding the misuse of cryptography.

The conceptual framework provided by our investigation (and documented in this thesis, particularly in Chapter 2, Section 2.3) represents a sound contribution to the current knowledge base for the emergent field of study of cryptographic software security. This framework can support developers willing to use cryptography when developing secure software systems as well as security experts when supporting development teams.

4.1 Future directions

The work is not done. In order to promote cryptographic software security, we have to keep its knowledge base as complete and up-to-date as possible, which implies that a constant effort of investigation should be conducted. As technology keeps evolving and human behavior has greater influence on (secure) software development, we see that experiments with actual developers regarding cryptography misuse in specific technologies is a fertile area of research.

Specially, we noticed an increasing interest in evaluating the usability of security APIs in general [55] and of cryptographic APIs in particular [6, 60, 47, 75], from a developers' point of view, instead of the evaluation of cryptography controls' usability from a final users' point of view [40, 48, 53, 100].

Also, we foresee a new generation of tools and techniques to better assist both developers in building cryptographic software, as well as cryptography experts in supporting development teams. Tools and techniques can evolve, roughly speaking, in two directions. First, as support for coding tasks (e.g., crypto APIs, security frameworks, early detection static analysis tools) and design activities (e.g., reference architectures, case tools, code generators, and best practices). Second, as support for verification activities in late detection of code-level issues (either static or dynamic detection) as well as in supporting the work of reviewers and auditors.

In the first direction, regarding cryptography misuse in coding tasks, there are opportunities for new APIs and frameworks showing alternative abstractions for cryptographic services, as well as (functional) test suites for crypto APIs and security frameworks, in order to compare them against a (predefined) baseline specification of acceptable behavior. Research in this direction is still in the beginning and concerns only correct usage of cryptographic algorithms. A few pioneering works are described below.

The NaCl crypto library [1, 16] was one of the first attempts to offer alternative abstractions for cryptographic services. NaCl was studied in the first usability evaluation of crypto APIs [6], showing promising results in simplifying to developers the concepts of public-key cryptography, despite its apparent difficulty when compared to other crypto libraries.

The Project Wycheproof [17] is the first attempt to provide publicly available unit tests for cryptographic libraries. These tests focus on implementation defects of cryptographic algorithms that can be detected by exercising the cryptographic API with appropriate parameters that expose the defect. Although not interested in cryptography misuse by application developers, Project Wycheproof uses negative unit tests to identify potential defects in algorithms' internals that can be accidentally exposed by application developers or exploited by adversaries. This approach is distinct from our previous works [35] in using test vectors as simple user stories and positive tests for validating implementations of cryptographic algorithms.

Regarding cryptography misuse in design activities, this investigation showed that there is important research being conducted in this direction. However, most of research is exploratory, mapping what is crypto misuse and how it can be detected by prototype tools. For instance, OpenCCE [14] is a prototype static analysis tool that also generates code for simple use cases of cryptography.

Due to the success we achieved in using unsupervised learning techniques (for association and clustering) to investigate cryptography misuse, we see as promising the use of other machine learning techniques as a supporting technology present in the next generation of tools. However, we warn that there is a need for techniques for automating the analysis of source code in order to identify crypto misuse in the first place, before using other machine learning techniques to find relevant relations among instances of crypto misuse.

In the second direction, regarding tools for detection of cryptography misuse, the definition of a **standardized** grounding truth against which such tools could be compared should be a first-order priority, because the current lack of standards resulted in non-overlapping tools and no reference baseline for what these tools should detect at a minimum. We found that there are important efforts being conducted to benchmark static analysis tools for security issues in general [80, 77, 44]. However, none of these works covered cryptography issues in depth. In this regard, our own research, the first practical evaluation of static analysis tools for cryptography, can contribute to strengthen the efforts for benchmarking vulnerability detection tools for cryptography.

Also, there are research opportunities in investigating better ways to assist reviewers and auditors to identify design flaws and insecure architectural choices regarding cryptography misuse in large software systems and complex architectures. We saw that most

effort is directed to code-level cryptography misuse, while design-level misuse (for instance, insecure composition of cryptographic services) and flawed architectures (for instance, insecure key management in distributed systems) are barely mentioned. Our own research found that the automated detection of cryptography misuse is far from being understood by current technology, presenting many blind spots in important areas, such as public-key cryptography.

4.2 Final message

We finish this text with a few words about empathy and multidisciplinary teams. This research would never be possible without the willing force to combine efforts and work together with non experts (experts in other fields), often putting ourselves in their shoes in order to (try to) understand why the obvious, from our point of view, is not that so to others. Cryptography, in spite of being full of wonderful mathematical constructs, is tough to developers. Thus, we had to learn how to smooth it for them, before they could handle it more easily.

We (software engineers, security experts, and cryptologists) are all allies in the ultimate goal of building secure software, despite all the uncertainty of current trends for fast-changing technology, agile development efforts, loosely defined user requirements, and increasingly dangerous threats.

When presenting our research to software engineering researchers, we were glad to see them getting surprised by just realizing that cryptography is not only that arcane science responsible for security features embedded into pluggable components, whose correctness is taken for granted. In fact, we received quite positive feedback about exposing cryptography misuse from a software engineering point of view and bringing developers to the loop of cryptographic software security.

Between cryptologists, who devise novel crypto systems, and users, who demand new security requirements, there are developers, who actually build applications. Software development is a human activity, highly influenced by social contexts. With the widespread use of cryptographic services, we (cryptography experts) have been pushed to software development. Today, it is not enough (although extremely necessary) to securely implement cryptography algorithms, pack them in executable binaries and release them in spartan APIs, intended to be developer-proof artifacts. We have to adapt ourselves and avoid the adversarial attitude against developers, trying to produce developer-friendly artifacts and tools.

This thesis emphasizes four complementary aspects that can help developers in systematically handle cryptography in the development of secure software and avoid cryptography misuse. First, a methodology for developing secure cryptographic software directs the infusion of cryptographic security into software functionality, from requirement identification to testing. Second, a classification of cryptography misuse for software security offers a taxonomy to locate actual instances of cryptography misuse in relation to each other as well as structures checklists of avoidable coding pitfalls in crypto software programming. Third, a reference architecture for cryptographic software helps to design a

layered stack of software abstractions, in which upper layers impose constraints that limit the behavior of lower layers, separating the concerns of distinct components of crypto systems and reducing the opportunities for cryptography misuse. Fourth, a set of guidelines on how to use static analysis tools to detect cryptography misuse helps to understand the limitations of these tools and still get some benefit when applying actual tools in specific development contexts, despite tools' imprecision and incompleteness.

Finally, this thesis also was an exercise in empathy that led us to figure out a new role in secure software development, the **cryptographic architect**. This new role is interested in the correct use of cryptographic features, as well as in the **safe** composition of cryptographic components, possibly at different levels of software abstractions (e.g., code and design), focusing attention on the architecture of cryptographic software, and considering not only the (social) context in which cryptography is applied, but also the forces influencing its proper use.

Bibliography

[1] NaCl: Networking and Cryptography library.

[2] The CRIME attack, 2012.

[3] The DROWN attack, 2016.

[4] The Logjam Attack and Weak Diffie-Hellman, 2016.

[5] Michel Abdalla, Benedikt Gierlichs, Kenneth G Paterson, Vincent Rijmen, Ahmad-Reza Sadeghi, Nigel P Smart, Martijn Stam, Michael Ward, Bogdan Warinschi, and Gaven Watson. Algorithms, Key Size and Protocols. Technical report, 2014.

[6] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*, 2017.

[7] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, and Others. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17. ACM, 2015.

[8] Eman Salem Alashwali. Cryptographic vulnerabilities in real-life web servers. In *Third International Conference on Communications and Information Technology (ICCIT)*, pages 6–11. Ieee, jun 2013.

[9] Nadhem AlFardan and Kenny Paterson. Lucky Thirteen Attack, 2013.

[10] Thijs Alkemade. Piercing Through WhatsApp's Encryption, 2013.

[11] Julia H. Allen, Sean Barnum, Robert J. Ellison, Gary McGraw, and Nancy R. Mead. *Software Security Engineering: A Guide for Project Managers (The SEI Series in Software Engineering)*. SEI Series in Software Engineering. Addison-Wesley Professional, 1 edition, 2008.

[12] Ross Anderson. Why cryptosystems fail. *Proceedings of the 1st ACM conference on Computer and communications security*, pages 215–227, 1993.

[13] Ross Anderson. *Security engineering*. 2008.

[14] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. Towards Secure Integration of Cryptographic Software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 1–13, New York, NY, USA, 2015. ACM.

[15] Taimur Aslam, Ivan Krsul, and Eugene H Spafford. Use of A Taxonomy of Security Faults. 1996.

[16] DJ Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. *Progress in Cryptology – LATINCRYPT 2012 (LNCS)*, 7533:159–176, 2012.

[17] Daniel Bleichenbacher, Thai Duong, Emilia Kasper, and Quan Nguyen. Project Wycheproof - Scaling crypto testing. In *Real World Crypto Symposium*, New York, USA, 2017.

[18] Joppe W Bos, J Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In *Financial Cryptography and Data Security*, pages 157–175. Springer, 2014.

[19] Pierre Bourque and Richard Fairley, editors. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society, version 3. edition, 2014.

[20] A Braga, C Rubira, and R Dahab. Tropyc: A pattern language for cryptographic object-oriented software, Chapter 16 in Pattern Languages of Program Design 4 (N. Harrison, B. Foote, and. In *Also in Procs. of PLoP*. 1999.

[21] Alexandre Braga. Cryptography Misuse Source Code Repository (crypto good use samples), 2017.

[22] Alexandre Braga. Cryptography Misuse Source Code Repository (crypto misuse instances), 2017.

[23] Alexandre Braga. Search in Google Play for cryptographic apps, 2017.

[24] Alexandre Braga and Alfredo Colito. Adding Secure Deletion to an Encrypted File System on Android Smartphones. In *The 8th Inter. Conf. on Emerging Security Information, Systems and Technologies*, pages 106–110, nov 2014.

[25] Alexandre Braga and Ricardo Dahab. A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software. In *XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg 2015)*, pages 30–43, Florianópolis, SC, Brazil, 2015.

[26] Alexandre Braga and Ricardo Dahab. Introdução à Criptografia para Programadores: Evitando Maus Usos da Criptografia em Sistemas de Software. In *Caderno de minicursos do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2015*, pages 1–50. 2015.

[27] Alexandre Braga and Ricardo Dahab. Mining Cryptography Misuse in Online Forums. In *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), International Workshop on Human and Social Aspect of Software Quality*, pages 143–150, 2016.

[28] Alexandre Braga and Ricardo Dahab. Towards a Methodology for the Development of Secure Cryptographic Software. In *The 2nd International Conference on Software Security and Assurance (ICSSA 2016)*, 2016.

[29] Alexandre Braga and Ricardo Dahab. A Longitudinal and Retrospective Study on How Developers Misuse Cryptography in Online Communities. In *XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg'17)*, Brasília, DF, Brazil, 2017.

[30] Alexandre Braga and Ricardo Dahab. Understanding the Field of Cryptographic Software Security. *Journal of Information and Software Technology*, 2017.

[31] Alexandre Braga, Ricardo Dahab, Nuno Antunes, Nuno Laranjeiro, and Marco Vieira. Practical Evaluation of Static Code Analysis Tools for Cryptography: Benchmarking Method and Case Study. In *The 28th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017.

[32] Alexandre Braga and Eduardo Morais. Implementation Issues in the Construction of Standard and Non-Standard Cryptography on Android Devices. In *The 8th International Conference on Emerging Security Information, Systems and Technologies*, pages 144–150, nov 2014.

[33] Alexandre Braga and Erick Nascimento. Portability evaluation of cryptographic libraries on android smartphones. pages 459–469. Springer, 2012.

[34] Alexandre Braga and Daniela Schwab. Design Issues in the Construction of a Cryptographically Secure Instant Message Service for Android Smartphones. In *The 8th Inter. Conf. on Emerging Security Information, Systems and Technologies*, pages 7–13, nov 2014.

[35] Alexandre Braga and Daniela Schwab. The Use of Acceptance Test-Driven Development to Improve Reliability in the Construction of Cryptographic Software. In *The 9th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2015).*, 2015.

[36] Alexandre Braga, Daniela Schwab, Eduardo Morais, Romulo Neto, and André Vannucci. Integrated Technologies for Communication Security and Secure Deletion on Android Smartphones. *International Journal On Advances in Security*, 8(1&2):28–47, 2015.

[37] Alexandre Braga, Romulo Zanco Neto, André Vannucci, and Ricardo Hiramatsu. Implementation Issues in the Construction of an Application Framework for Secure SMS Messages on Android Smartphones. In *The 9th Inter. Conf. on Emerging Security Information, Systems and Technologies*, pages 67–73. IARIA, 2015.

[38] Alexia Chatzikonstantinou, Christoforos Ntantogian, Christos Xenakis, and Georgios Karopoulos. Evaluation of Cryptography Usage in Android Applications. *9th EAI International Conference on Bio-inspired Information and Communications Technologies*, 2015.

[39] Brian Chess and Jacob West. *Secure programming with static analysis*. 2007.

[40] Sandy Clark, Travis Goodspeed, Perry Metzger, Zachary Wasserman, Kevin Xu, and Matt Blaze. Why (Special Agent) Johnny (Still) Can'T Encrypt: A Security Analysis of the APCO Project 25 Two-way Radio System. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, page 4, Berkeley, CA, USA, 2011. USENIX Association.

[41] Codenomecon. The Heartbleed Bug, 2014.

[42] Nicolas Courtois. Attack on ECDSA in bitcoin, 2014.

[43] CYBSI. Avoiding The Top 10 Software Security Design Flaws, 2014.

[44] Gabriel Díaz and Juan Ramón Bermejo. Static analysis of source code security: Assessment of tools against SAMATE tests. *Information and Software Technology*, 55(8):1462–1476, 2013.

[45] Thai Duong and Juliano Rizzo. Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET. *EEE Symposium on Security and Privacy*, pages 481–489, may 2011.

[46] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. *ACM SIGSAC conference on Computer & comm. security (CCS'13)*, pages 73–84, 2013.

[47] Ksenia Ermoshina, Harry Halpin, and Francesca Musiani. Can Johnny Build a Protocol? Co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols. 2017.

[48] Sascha Fahl, Marian Harbach, and Thomas Muders. Helping Johnny 2.0 to encrypt his Facebook conversations. In *Proceedings of the Eighth ...*, 2012.

[49] Sascha Fahl, Marian Harbach, and Thomas Muders. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *ACM conference on Computer and communications security*, pages 50–61, 2012.

[50] N Ferguson, B Schneier, and T Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley, 2011.

[51] Frost&Sullivan and ISC2. The 2015 (ISC)2 Global Information Security Workforce Study, 2015.

[52] J. Gajrani, M. Tripathi, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan. spectra: A precise framework for analyzing cryptographic vulnerabilities in android apps. In *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 854–860, Jan 2017.

[53] SL Garfinkel and RC Miller. Johnny 2: a user test of key continuity management with S/MIME and Outlook Express. In *Proceedings of the 2005 symposium on Usable ...*, pages 13–24, 2005.

[54] Martin Georgiev, S Iyengar, and Suman Jana. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*, pages 38–49, 2012.

[55] Matthew Green and Matthew Smith. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security and Privacy*, 14(5):40–46, 2016.

[56] Peter Gutmann. Lessons Learned in Implementing and Deploying Crypto Software. *Usenix Security Symposium*, 2002.

[57] Shon Harris. *CISSP All-in-One Exam Guide, 6th Edition*. All-in-One. McGraw-Hill Education, 2012.

[58] M Howard, D LeBlanc, and J Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill Education, 2009.

[59] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA, 2006.

[60] Soumya Indela, Mukul Kulkarni, Kartik Nayak, and Tudor Dumitraş. Helping Johnny encrypt: toward semantic interfaces for cryptographic frameworks. *Proc. of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2016*, pages 180–196, 2016.

[61] Tibor Jager and Juraj Somorovsky. How to break XML encryption. *Proceedings of the 18th ACM conference on Computer and communications security - CCS '11*, page 413, 2011.

[62] Pascal Junod. Towards Developer-Proof Cryptography. Technical report, EPFL, Summer Research Institute, Lausanne, Switzerland, 2016.

[63] Jonathan Katz and Yehuda Lindell. Introduction to Modern Cryptography. 2006.

[64] Muhammad Umair Ahmed Khan and Mohammed Zulkernine. On Selecting Appropriate Development Processes and Requirements Engineering Methods for Secure Software. *33rd Annual IEEE International Computer Software and Applications Conference*, pages 353–358, 2009.

[65] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). *Advances in Cryptology—CRYPTO 2001*, 2001.

[66] Carl E Landwehr, Alan R Bull, John P McDermott, and William S Choi. A Taxonomy of Computer Program Security Flaws. *ACM Comput. Surv.*, 26(3):211–254, 1994.

[67] Adam Langley. Apple's SSL/TLS "Goto fail" bug, 2014.

[68] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why Does Cryptographic Software Fail?: A Case Study and Open Problems. In *5th Asia-Pacific Workshop on Systems*, APSys '14, pages 7:1—-7:7, New York, NY, USA, 2014. ACM.

[69] Nancy Leveson. *Engineering a Safer World Systems Thinking Applied to Safety*. MIT Press, 2011.

[70] V Gayoso Mart and L Hern. Implementing ECC with Java Standard Edition 7. *International Journal of Computer Science and Artificial Intelligence*, 3(4):134–142, 2013.

[71] Gary McGraw. *Software Security: Building Security in*. 2006.

[72] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[73] Mark S Merkow and Lakshmikanth Raghavan. *Secure and Resilient Software Development*. CRC Press, 2010.

[74] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. The POODLE attack, 2014.

[75] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. "Jumping Through Hoops": Why do Java Developers Struggle With Cryptography APIs? *The 38th International Conference on Software Engineering*, 2016.

[76] NIST. Cryptographic Module Validation Program (CMVP).

[77] NIST. Software Assurance Metrics And Tool Evaluation (SAMATE).

[78] NIST. Security Requirements for Cryptographic Modules (FIPS PUB 140-2), 2001.

[79] OWASP. Cryptographic Storage Cheat Sheet.

[80] OWASP. OWASP Benchmark Project.

[81] OWASP. OWASP Testing Project, 2015.

[82] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[83] PMI. *Software Extension to the PMBOK® Guide*. Project Management Institute, 5 edition, 2013.

[84] PMI and AgileAlliance, editors. *Agile Practice Guide*. Project Management Institute, 2017.

[85] Roger Pressman and Bruce Maxin. *Software Engineering: A Practitioner's Approach (8th ed.)*. 8th edition, 2014.

[86] Juliano Rizzo and T Duong. Practical padding oracle attacks. *Proc. of the 4th USENIX conf. on offensive technologies (2010)*, pages 1–9, 2010.

[87] Phillip Rogaway. The Moral Character of Cryptographic Work. Technical report, IACR-Cryptology ePrint Archive, 2015.

[88] Safecode. Fundamental Practices for Secure Software Development, 2011.

[89] B Schneier. Cryptographic design vulnerabilities. *Computer*, (September):29–33, 1998.

[90] Bruce Schneier. Designing Encryption Algorithms for Real People. *Proceedings of the 1994 workshop on New security paradigms.*, pages 98–101, 1994.

[91] Adi Shamir and Nicko Van Someren. Playing 'hide and seek'with stored keys. *Financial cryptography*, pages 1–9, 1999.

[92] Adam Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[93] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. Analysis on Password Protection in Android Applications. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on*, pages 504–507, nov 2014.

[94] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In *IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC)*, pages 75–80, 2014.

[95] skype.com. Does Skype use encryption?

[96] Nigel Smart, Vincent Rijmen, Martijn Stam, Bogdan Warinschi, and Gaven Watson. Study on cryptographic protocols. 2014.

[97] Ian Sommerville. *Software Engineering*. International Computer Science Series. Pearson, 2011.

[98] source.android.com. Android Encryption, 2017.

[99] Supremo Tribunal Federal. STF inicia audiência pública que discute bloqueio judicial do WhatsApp e Marco Civil da Internet, 2017.

[100] Michael Sweikata, Gary Watson, Charles Frank, Chris Christensen, and Yi Hu. The usability of end user cryptographic products. In *2009 Information Security Curriculum Development Conference on - InfoSecCD '09*, pages 55–59, New York, New York, USA, 2009. ACM Press.

[101] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[102] telegram.org. Telegram MTProto Mobile Protocol, 2017.

[103] US-CERT. The FREAK attack, 2015.

[104] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. 2001.

[105] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. Bug characteristics in blockchain systems: a large-scale empirical study. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 413–424. IEEE Press, 2017.

[106] Sam Weber, Paul A Karger, and Amit Paradkar. A software flaw taxonomy: aiming tools at security. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[107] whatsapp.com. WhatsApp Encryption Overview, 2017.