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.

©2015 SBC — Soc. Bras. de Computação

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.

 ©2015 SBC — Soc. Bras. de Computação

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.

 ©2015 SBC — Soc. Bras. de Computação

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].

 ©2015 SBC — Soc. Bras. de Computação

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.

 ©2015 SBC — Soc. Bras. de Computação

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

 ©2015 SBC — Soc. Bras. de Computação

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.

 ©2015 SBC — Soc. Bras. de Computação

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.

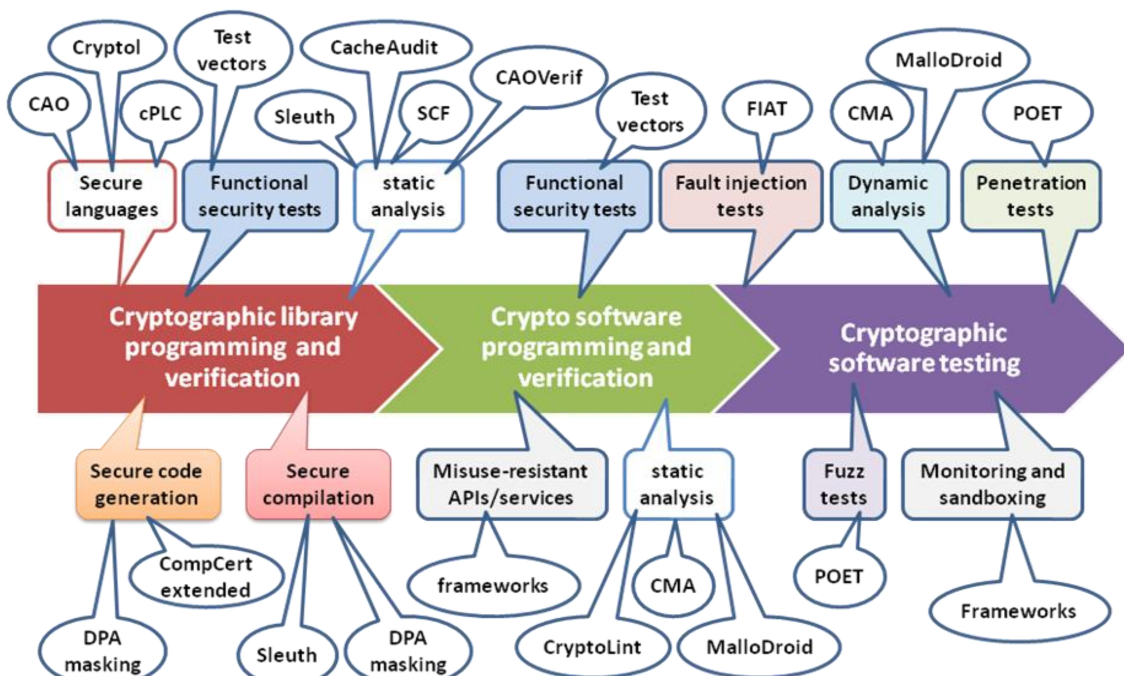

Figure 1: Automated tools in cryptographic software programming and testing.

©2015 SBC — Soc. Bras. de Computação

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

 ©2015 SBC — Soc. Bras. de Computação

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.

 ©2015 SBC — Soc. Bras. de Computação

[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.

 ©2015 SBC — Soc. Bras. de Computação

[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.

 ©2015 SBC — Soc. Bras. de Computação

[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.

 ©2015 SBC — Soc. Bras. de Computação