Why does cryptographic software fail? A case study and open problems

David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich
MIT CSAIL

Abstract

Mistakes in cryptographic software implementations often undermine the strong security guarantees offered by cryptography. This paper presents a systematic study of cryptographic vulnerabilities in practice, an examination of state-of-the-art techniques to prevent such vulnerabilities, and a discussion of open problems and possible future research directions. Our study covers 269 cryptographic vulnerabilities reported in the CVE database from January 2011 to May 2014. The results show that just 17% of the bugs are in cryptographic libraries (which often have devastating consequences), and the remaining 83% are misuses of cryptographic libraries by individual applications. We observe that preventing bugs in different parts of a system requires different techniques, and that no effective techniques exist to deal with certain classes of mistakes, such as weak key generation.

1 Introduction

Cryptographic algorithms and protocols are an important building block for a secure computer system. They provide confidentiality, integrity, and authentication based on solid mathematical foundations, and they are widely believed to provide strong security guarantees even against powerful adversaries like the NSA [23, 25]. However, turning mathematical equations into a working system is a difficult task and is where cryptographic systems usually fail [1]. Programmers have to faithfully implement algorithms, correctly interact with real-world input, choose appropriate parameters and configurations, and optimize for performance. A mistake in any of these steps can subvert the security of the entire system.

A recent example is Apple's "goto" bug in its SSL/TLS implementation, disclosed in February 2014, and shown in Figure 1. It is believed that a programmer accidentally added one redundant `goto` statement, probably through copying and pasting, effectively bypassing all certificate checks for SSL/TLS connections. This bug existed for over a year, during which millions, if not billions, of devices were vulnerable to man-in-the-middle attacks. Ironically, within two weeks a more serious "goto" bug was discovered in GnuTLS's certificate validation code; the GnuTLS implementation, as well as hundreds of software packages that depend on it, had been broken for more than ten years!

```
    if ((err = SSLHashSHA1.update(...)) != 0)
        goto fail;
        goto fail; /* BUG */
    if ((err = SSLHashSHA1.final(...)) != 0)
        goto fail;
    err = sslRawVerif(...);
    ...
fail:
    ...
    return err;
```

Figure 1: Apple's SSL/TLS `goto` bug (CVE-2014-1266), where an extra `goto` statement causes iOS and Mac devices to accept invalid certificates, making them susceptible to man-in-the-middle attacks.

Many pitfalls in cryptographic implementations are known to experts, and have been described in books [14], blog posts [21], and talks [15]. However, many of them focus on specific aspects of a cryptographic system, anecdotal evidence, or specific protocols and applications, making it difficult to draw broader conclusions about the kinds of mistakes that occur in practice.

This paper systematically investigates the mistakes that arise when implementing and using cryptography in real-world systems, and makes the following contributions.

The first contribution is an analysis of 269 vulnerabilities that were marked as "Cryptographic Issues" (CWE-310) in the CVE database [26] from January 2011 to May 2014. The analysis, presented in §2, classifies the vulnerabilities according to what kind of mistake made the system vulnerable, what part of the system contained the mistake, and the impact of the mistake on the system's security. We hope this analysis will help developers learn about common pitfalls and how to avoid them.

The second contribution is an examination of state-of-the-art techniques for preventing such vulnerabilities, such as testing, static analysis, formal verification, and better API design; §3 discusses these techniques in terms of how a developer can apply them when building a sys-

APSys '14, June 25–26, 2014, Beijing, China.

tem, and how well they address mistakes that occur in practice.

The third contribution of our paper is a discussion of open problems that remain in building secure cryptographic systems. In §4, we describe what problems remain unsolved in light of our analysis, and speculate on possible future research directions.

Our findings show that just 17% of mistakes occur in core cryptographic primitives and libraries; this code is typically developed by experts, and any mistakes in it affect many applications that use these libraries. The other 83% of mistakes are in individual applications; they typically involve the application misusing cryptographic libraries, and affect just that application's security. We find that different techniques are necessary to prevent mistakes at various layers of a system, from application bugs, to cryptographic protocols like SSL/TLS, to cryptographic primitives like Diffie-Hellman. Finally, we find that no effective techniques exist to handle certain classes of mistakes, such as poor key generation as a result of inadequate checks for weak keys or insufficient entropy.

2 Vulnerabilities

To assess the kinds of cryptographic vulnerabilities that occur in practice, we categorized 337 CVEs tagged "Cryptographic Issue" (CWE-310) from January 2011 to May 2014. We excluded 68 CVEs from our study: 42 bugs that cause denial of service but do not break the cryptographic intent of the software, 8 file permission errors, 6 user interface issues, and 12 CVEs with insufficient information. The remaining 269 CVEs are summarized in Figure 2.

The bold groupings in Figure 2 represent the impact of a vulnerability: direct disclosure of plaintext data; man-in-the-middle attacks that involve impersonating a server; brute-force attacks that involve guessing cryptographic keys; and side-channel attacks. The vulnerabilities that can lead to each of these impacts are indented in the first column, and we describe them in more detail in the rest of this section.

The other columns in Figure 2 represent the layers of a system that can contain a mistake. We consider three layers. "Primitive" corresponds to cryptographic primitives like AES, Diffie-Hellman, or El-Gamal; these are typically implemented behind an interface that makes it possible to switch between primitives (e.g., switching from DES to AES). "Protocol" corresponds to a cryptographic library that implements a cryptographic protocol like SSL/TLS or X.509, or uses the cryptographic primitives to implement a higher-level abstraction, such as a library for storing hashed passwords; this code should be written by cryptographic experts. Finally, "Application" refers to the rest of the application, which does not need to be written by cryptographic experts, but does need to use the cryptographic libraries appropriately.

Vulnerability type	Primitive	Protocol	Application
Plaintext disclosure			
Plaintext communication			38
Plaintext storage			32
Plaintext logging			9
Man-in-the-middle attacks			
Authentication logic error		10	22
Inadequate SSL cert. checks			42
Brute-force attacks			
Encryption logic error	1	4	
Weak encryption cipher		2	35
Weak keys	4	2	2
Hard-coded keys		1	25
Weak PRNG		2	2
Low PRNG seed entropy		1	16
PRNG seed reuse		4	
Side-channel attacks			
Timing	2	8	
Padding oracle		2	
Compression (CRIME)		2	
Memory disclosure		1	
Total	7	39	223

Figure 2: Categories of cryptographic vulnerabilities. Vulnerability types are grouped by their impacts (in bold text). Each number is a count of the CVEs in a category at the primitive, protocol, or application layer of cryptographic software.

2.1 Plaintext disclosure

A common error we found was forgetting to encrypt important data. Common examples include not using HTTPS for web logins (e.g., CVE-2014-1242) and storing passwords in plaintext (e.g., CVE-2013-7127). All of these bugs are application-specific, and do not involve cryptographic software itself. Their impact and solutions have been widely explored, such as in the context of taint tracking [28] and information flow control [29]. We do not discuss them further in the rest of this paper.

2.2 Man-in-the-middle attacks

Authenticated cryptography uses message authentication codes and digital signatures to prevent man-in-the-middle attackers from tampering with data and spoofing identities. Incorrect implementations or misuse of authenticated cryptography can fail to achieve this goal.

Authentication logic errors. One of the recent major authentication errors is CVE-2014-1266 in Apple iOS and OS X: as shown in Figure 1, a misplaced `goto` statement causes Apple devices to accept invalid signatures during a TLS key exchange. GnuTLS prior to version 3.2.12 also verifies TLS certificates incorrectly (CVE-2014-0092), affecting hundreds of packages that depend on it.

Inadequate checks. Even when the SSL/TLS library is implemented correctly, many applications perform inadequate checks on SSL/TLS certificates (e.g., CVE-2014-

```
const char *ptr = ...;
unsigned int tmp = 0;
...
tmp <<= 8;
tmp |= *ptr; /* FIX: tmp |= (unsigned char)*ptr;  */
```

Figure 3: A sign-extension bug in `crypt_blowfish` that leads to an encryption logic error with `0x80` characters (e.g., CVE-2011-2483).

1263, where `curl` does not check certificates for URLs that contain an IP address instead of a hostname) or perform no checks at all (e.g., CVE-2014-1967). In all of these cases, a man-in-the-middle attacker can craft a special certificate to spoof an SSL server and obtain sensitive information. Inadequate checks are not limited to SSL/TLS certificates; for instance, Android failed to correctly check signatures on APK files (CVE-2013-4787).

2.3 Brute-force attacks

This subsection describes two broad classes of mistakes that allow an adversary to compromise cryptography through brute-force guessing.

2.3.1 Low encryption strength

Encryption logic error. Logic errors in an encryption implementation can weaken its effectiveness. Figure 3 shows such an example, a signedness bug in `crypt_blowfish` that does not anticipate input that contains `0x80` characters with the most significant bit set (e.g., Unicode strings); this bug would make keys easier to crack. Particularly, `*ptr` is of type `char`, which is interpreted as a signed integer type on many platforms; converting the value to `unsigned int` is a sign-extension, which should have been a zero-extension. §3.2 will further discuss how this bug slipped through testing.

Weak encryption algorithms. The use of weak or obsolete cryptographic algorithms, such as MD4, MD5, SHA1, DES, and RC4, leads to systems that are susceptible to brute-force attacks. We believe that programmers do so often due to historical, out-of-date practices that were once considered secure, as well as insecure default configurations in the cryptography library they use.

For example, OpenAFS has been using DES for encryption (CVE-2013-4134), even though nowadays cracking DES keys is fast and cheap (under $100 within 24 hours even with publicly available services [20]). For another example, 65% of Android applications that use cryptography were found to use the default, insecure ECB mode for AES encryption [11].

Weak encryption keys. We found weak key vulnerabilities at each layer we investigated. At the primitive layer, many encryption algorithms have a known set of weak keys that weaken the resulting encryption. Implementations should avoid generating these weak keys. In CVE-2014-1491, Mozilla's NSS library allows weak public key values that break the security of Diffie-Hellman key exchange. At the protocol level, some implementations of SSL accept RSA keys that are shorter than 1024 bits (e.g., CVE-2012-0655).

Hard-coded encryption keys. Surprisingly many applications, even open-source ones, hard-code private encryption keys. This might be due to a misconception that compilation obfuscates constants that appear in the source code. Regardless, encryption with hard-coded keys provides little to no security benefit.

2.3.2 Insufficient randomness

Random number generation is crucial to cryptographic systems. For example, random numbers are used to generate unguessable secret keys. High-level cryptographic protocols, such as SSL, use random challenges during the authentication process to protect against replay attacks.

Ideally, cryptographic software should always use *true* random numbers to ensure those secret values are unguessable by adversaries. In reality, however, when sources of entropy are insufficient, many implementations will resort to pseudo-random number generators (PRNGs) to produce required random sequences. A PRNG is a function which takes a certain amount of true randomness, called a *seed*, and deterministically outputs a stream of bits which can be used as if they were random.

For a cryptographic system that uses a PRNG, its security will depend on two factors: the choice of PRNG algorithm and the choice of its seed. Weaknesses in either of them could result in insufficient randomness. We have encountered three common causes of insufficient randomness.

Weak PRNG. Cryptographic code should use a cryptographically secure PRNG (CSPRNG for short). CSPRNGs are designed so that predicting future bits reduces to hard problems in cryptography, such as inverting AES. Developers can easily mistake a statistical (not cryptographically secure) PRNG for a CSPRNG (e.g., CVE-2013-2803 and CVE-2013-6386). Statistical PRNGs, such as Mersenne Twister, the `srandom` function on Linux, or `Math.random` in many libraries, are great for games and Monte Carlo simulations, but are too predictable for cryptographic uses.

Predictable PRNG seeds. For a PRNG to be effective in a cryptographic setting, it must be seeded with truly random bits, say from `/dev/random`. A common error is to seed a PRNG with predictable values like the current time (e.g., CVE-2011-0766), process PID (e.g., CVE-2011-2190), or even a constant value (e.g., CVE-2013-6394). Some versions of Libc in OS X use uninitialized data as a seed. Reading uninitialized data is undefined behavior in C, so the code that seeds the PRNG could

be deleted by compiler optimizations [27]. Similarly, the infamous Debian SSL bug (CVE-2008-0166) was caused by a maintainer trying to suppress Valgrind warnings in code that read uninitialized data to seed a PRNG.

Seed reuse. To ensure that random numbers are independent across child processes, a PRNG should be reseeded in each child. For example, stunnel before 5.00 (CVE-2014-0016) and libssh before 0.6.3 (CVE-2014-0017) did not reseed the PRNG in forked child processes, so an attacker that knows the PRNG outputs of one process can predict the randomness of another child process. Earlier versions of Ruby had a similar issue (CVE-2011-2686).

2.4 Side-channel attacks

One noteworthy challenge of implementing cryptographic software is to defend against side-channels attacks, which break cryptography using information leaked through unintended sources, such as computation time, how data is padded or compressed, and memory contents. PolarSSL, for example, was vulnerable to a sophisticated timing attack on its implementation of the Montgomery multiplication algorithm (CVE-2013-5915 [2]). Applications can also be vulnerable to side-channel attacks [7], but our CVE dataset contained side-channel attack vulnerabilities only at the primitive and protocol layers.

3 Prevention techniques

This section examines existing techniques that can help prevent certain vulnerabilities in cryptographic implementations, and discusses the limitations of these techniques.

3.1 Design

Careful design of cryptographic primitives, protocols, and applications can help avoid some of the pitfalls described in the previous section, and thus make it easier to build correct, secure implementations.

One example primitive is Ed25519, a signature scheme designed to have no branches or memory addresses that depend on secret information [4]. This aspect of its design greatly simplifies the task of avoid certain side-channel vulnerabilities when implementing Ed25519. Better design of cryptographic primitives appears also promising for dealing with weak keys.

On the protocol side, Marchenko and Karp [16] describe how implementations of OpenSSH and OpenSSL can be structured to minimize the impact of software vulnerabilities in different parts of the implementation. This can protect cryptographic secrets in one part of the cryptographic library from bugs in other code. However, it cannot prevent most of the vulnerabilities described in the previous section, which arise from mistakes in the very code responsible for handling cryptographic secrets.

At the application level, high-level cryptography APIs that hide details like encryption algorithms, block cipher modes, and key lengths from programmers can prevent many of the mistakes that arise in application code. The Keyczar cryptographic library [10] takes this approach. Figure 4 shows how an application programmer encrypts and decrypts data using the Keyczar encryption API, which provides just two methods, `encrypt` and `decrypt`. The data is encrypted with a strong block cipher and the resulting ciphertext is signed and authenticated automatically. Furthermore, secret keys are decoupled from the code, preventing hard-coded key errors.

```
crypter = Crypter.Read(path_to_keys)
ciphertext = crypter.Encrypt("Secret message")
plaintext = crypter.Decrypt(ciphertext)
```

Figure 4: Python code for encrypting and decrypting data with the Keyczar API, which takes care of choosing appropriate configurations and parameters for the underlying cryptographic primitives.

Although better design can help avoid certain pitfalls, applying this advice can be difficult, especially in complex systems with many conflicting requirements, and may be incompatible with the use of existing primitives and protocols. Moreover, even implementations of well-designed systems can have mistakes. In the rest of this section, we look at techniques to prevent vulnerabilities in implementations based on existing designs.

3.2 Testing

Testing is a time-honored way of finding bugs in programs, and cryptographic software is no exception. For instance, the National Institute of Standards and Technology (NIST) has published test vectors for a number of cryptographic primitives [19]. Every implementation should pass these tests.

High code coverage is critical to the success of testing, and yet it is non-trivial to achieve, even for testing cryptographic primitives that have well-defined input and output. One interesting example is that many test vectors use 7-bit ASCII strings, such as English letters and digits, and thus would miss bugs that manifest themselves only with input that contains `0x80` characters (e.g., Unicode strings), as shown in Figure 3. This particular case has appeared multiple times in the past two decades, including CVE-2012-2143 (`crypt_des`), CVE-2011-2483 (`crypt_blowfish`), and the Blowfish bug in 1996 [18], where their test vectors did not consider `0x80` characters and failed to catch the corresponding bugs.

Testing protocols or applications for cryptographic bugs is difficult because bugs violate a complex invariant, rather than crash the program, and because triggering a bug may require a carefully constructed input, as in Apple's `goto` bug. As a result, protocol or application testing often involves ad-hoc test cases and regression test suites designed to prevent past mistakes.

3.3 Static analysis

Compilers and many bug-finding tools issue warnings about general programming errors. The extra `goto` in Apple's TLS implementation (Figure 1) caused a lot of unreachable code, which could have been caught by enabling the warning option `-Wunreachable-code` in GCC or Clang. For this reason, we recommend that all cryptographic software should be warning-free with the compiler's strictest settings. Most other cryptographic errors cannot be caught with the compiler alone.

Another set of bug-finding tools focus on cryptographic errors in applications. MalloDroid [13] performs static analysis to identify Android applications that do not properly check SSL certificates. CryptoLint [11] is another static analysis tool for Android applications that catches common cryptographic errors: hard-coded seeds, keys, and IVs, using AES in ECB mode, and using too few iterations for password-based encryption. We believe these tools can be extended to catch more classes of bugs, but more powerful techniques such as formal verification are needed to make strong guarantees about the security of primitive and protocol implementations.

3.4 Verification

There has been a rich literature of applying formal verification techniques to analyzing the security properties of cryptographic protocol specifications, such as Murφ [17] and PCL [9]. In this section we focus on verification techniques for checking *implementations*.

Unlike testing and static analysis, formal verification methods provide a formal proof of an abstract mathematical model of the implementation, and thus generally give stronger guarantees about its correctness.

Previous works use formal methods to verify both block ciphers and cryptographic hash algorithms [12, 24]. These tools usually rely on a SAT solver to decide the equivalence of two symbolic models—one extracted from the implementation and the other from the specification. One limitation of this approach is that it cannot handle inputs of variable length: to generate a model the SAT solver can accept, all loops in the program have to be unrolled.

The same principle can be applied to verifying implementations of cryptographic protocols. For example, the Crypto Verification Kit (CVK) from Microsoft Research translates protocols written in a subset of a functional language (F#) into ProVerif [6] input, in which the symbolic model can be proved. This technique is used to verify an implementation of TLS 1.0 [5]. CVK can also enforce assertion-based security properties, which are commonly seen in authorization policies [3], using refined type annotation given by the programmer.

Another strategy to secure systems with formal methods is to automatically transform a protocol specification into functional code, using a certified compiler [8]. Since the implementation is directly derived from the specification, the two are guaranteed to be equivalent.

4 Open problems

The analysis of prevention techniques in the previous section suggests that tackling bugs at each layer of the system requires a different approach, and that open problems remain at each of the layers, as we will now discuss.

4.1 Cryptographic primitives

Correctness proofs for implementations of cryptographic primitives often rely on SAT solvers, but this has two important limitations. First, SAT solvers are limited to loop-free constraints, and thus can prove correctness only for fixed-size inputs; proving a primitive's implementation correct for arbitrary inputs requires inductive reasoning about the code, which is more difficult to automate. Second, using a SAT solver to naïvely check the correctness of the entire implementation is not scalable, because the constraints become too large. Addressing this requires breaking up the implementation into smaller chunks, so that SAT solvers can solve smaller constraints, but finding such a partitioning and combining the sub-proofs to construct an overall proof is non-trivial.

Most of the current work on verifying primitives focuses on "common case" operations, such as encryption and decryption, or signing and verification. However, as we saw in Figure 2, bugs can arise in key generation, where insufficient checks for weak keys make the key susceptible to brute-force attacks. Checking for such mistakes requires a combination of mathematical understanding of what constitutes a weak key, as well as program analysis to determine if weak keys are correctly avoided.

Side-channel attacks can be present in implementations of cryptographic primitives, but current approaches deal with such attacks in an ad-hoc manner. Addressing side-channel attacks in a more principled way requires a precise model of additional information revealed by a system, such as code execution time and packet sizes.

Analyzing implementations of cryptographic primitives can be difficult because they often rely on low-level, architecture-specific optimizations to improve performance. These optimizations make it difficult to extract the mathematical structure from the underlying assembly code. One solution may be to construct a precise model for assembly code; another approach is to synthesize efficient assembly code for a given architecture from a higher-level and easier-to-analyze specification.

4.2 Cryptographic protocols

One open problem for verifying implementations of cryptographic protocols lies in dealing with composition: for example, the CRIME attack [22] exploits systems that

compress plaintext data before encryption, and that allow an adversary to influence plaintext data. Current work on verifying SSL/TLS implementations does not yet scale to such issues. Applying ideas from PCL [9] to analyzing implementations may be a promising approach.

Another open problem lies in better testing of existing implementations that are difficult to verify outright (e.g., because they are not written in a formal specification language). Symbolic execution is a powerful approach for synthesizing test cases with high coverage. However, to symbolically execute protocol implementations would require extending the symbolic execution system to know about cryptographic primitives: how to synthesize inputs that pass cryptographic checks like signatures or MACs, and how to transform inputs that pass through encryption or decryption. Testing also requires specifying bug conditions, since symbolic execution systems typically expect crashes to signify bugs. On the other hand, if verification can be made practical for a fully featured SSL/TLS implementation, new testing techniques may not be needed.

4.3 Applications

The prevalence of application-level mistakes in using cryptographic libraries demonstrates the need for better interfaces that are less prone to misuse, along the lines of Keyczar [10]. We found several classes of mistakes that application developers make, which should be addressed in library interfaces. First, interfaces should default to providing strong ciphers, ensure the use of strong PRNGs, and help developers provide sufficient seed entropy and avoid hard-coding keys. Second, interfaces should help developers properly authenticate messages and check certificates. Third, interfaces or tools may be able to help developers avoid "plaintext disclosure" problems where sensitive data is not encrypted in the first place.

Alternatively, bug-finding tools that look for incorrect uses of a cryptographic library, akin to CryptoLint [11], can be extended to cover a wider range of interface misuses that we uncovered in Figure 2.

5 Discussion

The case study presented in the previous sections leads us to believe that new techniques for preventing protocol-level bugs would have the most significant impact on cryptographic software in general. Application bugs, although common, have localized impact and many of these bugs can be caught with static analysis tools. There was only one logical error at the primitive level, suggesting that new techniques to verify primitives would not greatly increase security. The weak key bugs in primitives are worth addressing, but the nature of these bugs is that they are rarely triggered in practice. Finally, the remaining protocol-level bugs have wide impact and are not easily preventable with existing techniques.

It is also important to note that our categorization of CVEs presented in §2 is subjective. For some CVEs where source code was unavailable, we had to guess, based on the CVE's summary, if an error was in an application's use of a protocol or in its implementation of a protocol. Furthermore, our case study excludes certain bug classes, such as buffer overflow errors, that affect cryptographic software but do not fall under "Cryptographic Issues" (CWE-310). For example, the recent Heartbleed bug (CVE-2014-0160) falls under "Buffer Errors" (CWE-119), so it was not included in our analysis. There is a myriad of work on preventing such bugs.

6 Conclusion

This paper's analysis of mistakes in systems that implement and use cryptography demonstrates that 83% of the bugs are in applications that misuse cryptographic libraries, and just 17% of the bugs are in the cryptographic libraries themselves. We find that different techniques are needed to tackle bugs in cryptographic primitives, protocols, and application-level code. Existing techniques can handle only a subset of mistakes we observe in practice, and we propose several open problems to tackle in future research, including weak key generation, protocol testing, and proving the correctness of cryptographic primitives for arbitrary inputs.

Acknowledgments

Thanks to the anonymous reviewers for their feedback. This work was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

[1] R. Anderson. Why cryptosystems fail. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS)*, pages 215–227, Fairfax, VA, Nov. 1993.

[2] C. Arnaud and P.-A. Fouque. Timing attack against protected RSA-CRT implementation used in PolarSSL. In *Proceedings of the Cryptographer's Track at RSA Conference (CT-RSA)*, pages 18–33, San Francisco, CA, Feb. 2013.

[3] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems*, 33(2):8, 2011.

[4] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sept. 2012.

[5] K. Bhargavan, R. Corin, C. Fournet, and E. Zalinescu. Cryptographically verified implementations for TLS. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 459–468, Alexandria, VA, Oct. 2008.

[6] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 82–96, June 2001.

[7] A. Bortz, D. Boneh, and P. Nandy. Exposing private information by timing web applications. In *Proceedings of the 16th International Conference on World Wide Web*, pages 621–628, May 2007.

[8] D. Cadé and B. Blanchet. From computationally-proved protocol specifications to implementations. In *Proceedings of the Seventh International Conference on on Availability, Reliability and Security (ARES)*, pages 65–74, 2012.

[9] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172:311–358, Apr. 2007.

[10] A. Dey and S. Weis. Keyczar: A cryptographic toolkit, 2008. http://www.keyczar.org/.

[11] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 73–84, Berlin, Germany, Nov. 2013.

[12] L. Erkök and J. Matthews. Pragmatic equivalence and safety checking in Cryptol. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, pages 73–82, 2009.

[13] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 50–61, Raleigh, NC, Oct. 2012.

[14] N. Ferguson, B. Schneier, and T. Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley, Mar. 2010.

[15] A. Langley. HTTPS: things that bit us, things we fixed and things that are waiting in the grass. Workshop on Real-World Cryptography, Jan. 2013. https://www.imperialviolet.org/2013/01/13/rwc03.html.

[16] P. Marchenko and B. Karp. Structuring protocol implementations to protect sensitive data. In *Proceedings of the 19th USENIX Security Symposium*, pages 47–62, Washington, DC, Aug. 2010.

[17] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *Proceedings of the 18th IEEE Symposium on Security and Privacy*, pages 141–151, Oakland, CA, May 1997.

[18] M. Morgan. Blowfish can be cracked! (fix included...), July 1996. https://www.schneier.com/blowfish-bug.txt.

[19] National Institute of Standards and Technology. Cryptographic algorithm validation program. http://csrc.nist.gov/groups/STM/cavp/.

[20] OpenAFS. Brute force DES attack permits compromise of AFS cell (CVE-2013-4134), July 2013. http://www.openafs.org/pages/security/OPENAFS-SA-2013-003.txt.

[21] C. Percival. Cryptographic right answers, June 2009. http://www.daemonology.net/blog/2009-06-11-cryptographic-right-answers.html.

[22] J. Rizzo and T. Duong. The CRIME attack. ekoparty Security Conference, Sept. 2012. http://www.ekoparty.org/archive/2012/CRIME_ekoparty2012.pdf.

[23] B. Schneier. NSA surveillance: A guide to staying secure, Sept. 2013. https://www.schneier.com/essay-450.html.

[24] E. W. Smith and D. L. Dill. Automatic formal verification of block cipher implementations. In *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Portland, OR, Nov. 2008.

[25] E. Snowden. NSA whistleblower answers reader questions, June 2013. http://www.theguardian.com/world/2013/jun/17/edward-snowden-nsa-files-whistleblower.

[26] The MITRE Corporation. Common vulnerabilities and exposures (CVE). http://cve.mitre.org/.

[27] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the 3rd Asia-Pacific Workshop on Systems*, Seoul, South Korea, July 2012.

[28] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 291–304, Big Sky, MT, Oct. 2009.

[29] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, Seattle, WA, Nov. 2006.