From Theory to Code: Identifying Logical Flaws in Cryptographic Implementations in C/C++*

Sazzadur Rahaman, Haipeng Cai, Omar Chowdhury and Danfeng (Daphne) Yao, *Senior Member, IEEE*

Abstract—Cryptographic protocols are often expected to be provably secure. However, this security guarantee often falls short in practice due to various implementation flaws. We propose a new paradigm called *cryptographic program analysis (CPA)* which prescribes the use of program analysis to detect these implementation flaws at compile time. The principal insight of the CPA is that many of these flaws in cryptographic implementations can be mapped to the violation of *meta-level properties* of implementations. A program property that is necessary to realize a cryptographic property is referred to as meta-level property. We show that violations of these meta-level properties can be identified at compile-time that can serve as sufficient evidence of the encompassing flaws. We investigated existing literature on cryptographic implementation flaws and derived 25 corresponding meta-level properties. To instantiate the abstract paradigm of CPA, we develop a specification language based on deterministic finite automaton (DFA) and show that most of the meta-level properties can be expressed in terms of our language. We then develop a tool called TAINTCRYPT which uses static taint analysis to identify meta-level property violations of C/C++ cryptographic implementations at compile-time. We demonstrate the efficacy of TAINTCRYPT by analyzing open-source C/C++ cryptographic libraries (e.g., OpenSSL) and observe that TAINTCRYPT could have helped to avoid several high-profile flaws. We also evaluated TAINTCRYPT on 5 popular applications and libraries, which generated new security insights. The experimental evaluation on large-scale projects indicates the scalability of our approach.

Index Terms—Cryptographic code; information flow analysis; static tainting; security vulnerability

1 INTRODUCTION

Cryptographic protocols/constructs are often used as the building block for providing robust security guarantees in many applications (e.g., HTTPS [2], DNSSEC [3], SMTP-over-TLS [4]). While implementing or employing these cryptographic protocols, one hopes to replicate the security guarantees provided by their theoretical cryptographic counterparts, that have been proven to be secure. This seemingly straightforward goal of implementing applications with provably secure guarantees, however, is often unaccomplished as evident in the recent high-profile outbreaks of cryptography-related vulnerabilities in widely used network libraries and tools (e.g., heartbleed vulnerabilities in OpenSSL [5] and seed leaking in Juniper Network [6]).

The lack of provable security guarantees in applications relying on cryptography can be often attributed to a combination of the following reasons: (1) The application uses a vulnerable cryptographic library or an insecure cryptographic construct/parameter (e.g., MD5 hash function); (2) A cryptographic construct is used without satisfying its required precondition (e.g., initialization vector not being random); (3) The correct APIs of the underlying cryptographic library are not invoked at all, not invoked in the prescribed order, or invoked with improper arguments (e.g., hostname validation is not performed after X.509 certificate chain validation); (4) The application suffers from logical/run-time vulnerabilities (e.g., buffer overflow). The impact of such insecurity for a critical application can affect millions of devices that execute the vulnerable implementation, potentially rendering millions of users vulnerable to adversarial attacks resulting in the loss of user privacy, vendor reputation, or even financial loss. *The main objective of the paper is to develop techniques for aiding developers to avoid such pitfalls in their applications.*

This paper contributes to this overarching vision by presenting a new paradigm called the *cryptographic program analysis* (CPA) which prescribes the use of program analysis approaches to develop compile-time insecurity checking and security enhancing solutions. Most of the existing work in this domain either focus on precisely detecting cryptographic API misuses by the applications [7], [8], [9], [10], [11], [12] or identifying protocol-specific vulnerabilities in the cryptographic libraries [13], [14], [15], [16], [17], [18]. These relevant efforts, however, leave the void of not assisting developers to avoid other kinds of pitfalls, for instance, their use of insecure cryptographic constructs (e.g., ECB mode in symmetric ciphers) or parameters (e.g., RSA public-exponent 3 [19]).

The key insight that enables CPA to effectively aid developers to have a robust implementation is that many of the aforementioned pitfalls can be mapped to the violations of *meta-level properties* of the implementations. A program property that is necessary to realize a cryptographic property is referred to as meta-level property. The violations of the meta-level properties cannot only be checked during compile-time but also their violations can serve as sufficient evidence of the cryptographic flaws they encompass. For explaining meta-level properties, let us take a fictitious application that processes commands from a client. For ensuring the integrity of the submitted command, it uses a 8-byte message authentication code (MAC) scheme. Let us also assume that the MAC scheme enjoys the desired security property

A preliminary version of the work appeared in the Proceedings of the IEEE Secure Development Conference, 2017 [1].

- *Sazzadur Rahaman and Danfeng (Daphne) Yao are with the Department of Computer Science, Virginia Tech. Haipeng Cai is affiliated with the School of Electrical Engineering and Computer Science, Washington State University. Omar Chowdhury is affiliated with the Department of Computer Science, The University of Iowa.*
E-mail: sazzad14@vt.edu, hcai@eecs.wsu.edu, omar-chowdhury@uiowa.edu, danfeng@vt.edu.

of *resistance against existential forgery* [20].

During its execution, whenever the application receives a plaintext command m and its MAC rmac_k^m, before processing the command m it checks the validity of the MAC rmac_k^m. When the MAC verification fails, it returns MAC_FAIL warning message; otherwise, it returns OK. For verifying the MAC, it first constructs the MAC of m using its key k. Suppose the constructed MAC is cmac_k^m. It then checks to see whether $\mathsf{cmac}_k^m \overset{?}{=} \mathsf{rmac}_k^m$ by comparing each byte of cmac_k^m with rmac_k^m; halting the comparison by sending MAC_FAIL when the first mismatch is observed. It is evident that through the observation of the response message and the timing of the received response message,

An dversary—who does not know the cryptographic key—can forge the MAC of a new message with only 8×256 attempts instead of 256^8. Based on prior work by [21], [22], [23], we speculate that this flaw can be easily mapped to the following meta-level property violation: "*No early termination during the comparison of cryptographic payloads*". In the similar vein, the infamous Bleichenbacher's padding oracle attack against RSA can be mounted due to the violation of another meta-level property: "*The same, generic error message should be sent whenever the protocol experiences an error condition*".

Many of the meta-level properties can be specific to cryptographic constructs/protocols. To enable the specification of such meta-level properties, we provide a deterministic finite automaton (DFA) based language. We also develop a tool dubbed TAINTCRYPT that leverages static information flow analysis to identify the violations of meta-level properties in C/C++ implementations. Our static analysis is both path- and context-sensitive, hence capable of enforcing a rich set of cryptographic properties precisely (i.e., small false positives).

Our work targets and addresses the fundamental challenge of mapping theoretical cryptographic concepts to practical code structures and security-related behavioral properties, which can potentially enable a wide range of code-based security analysis for cryptographic software. This work thus serves as an essential first step towards performing systematic, automated analyses of cryptographic libraries and their applications of millions of lines of code. Although static information flow analysis itself has been studied as a general methodology for reasoning about cryptographic code security [21], [22], [23], it remains untapped how this general technique can be leveraged to build a unified tool to detect a wide range of cryptographic vulnerabilities.

Contributions: In summary, this paper makes the following contributions:

- We conducted an in-depth exploratory study of code-level security vulnerabilities in cryptographic programs, which resulted in a taxonomy of 25 classes of exploitable vulnerabilities in cryptographic implementations that boil down to 12 distinct types of security attacks. Our exploratory study is based on (1) surveying the literature of existing cryptographic attacks; (2) observing the change-logs of OpenSSLs releases (Section 3). The purpose is to cover as many interesting attacks as possible within one program analysis tool so that developers can routinely use this tool to screen their code.
- We derived 25 enforceable rules (meta-level properties) from our vulnerability study and taxonomy, which address 6 out of the total of 12 security attacks identified. We further showed that static analysis can be used for 23 of these rules to capture the sufficient condition for proving if a property holds or not.
- We identified compile-time security checking of cryptographic implementations as an unexplored problem in software security and proposed a deterministic finite automaton (DFA) based language to express meta-level cryptographic properties that can be statically checked using static analysis. Further, we demonstrated our technique by developing a tool named TAINTCRYPT that enforces 15 security rules we derived from our exploratory study.
- We implemented TAINTCRYPT for C/C++ programs as a practical tool based on LLVM and used the tool to evaluate our CPA technique against real-world cryptographic software. We demonstrated the effectiveness and efficiency of our technique and thus showed how static information flow analysis can be exploited to diagnose a large variety of cryptographic vulnerabilities in large-scale libraries like OpenSSL and critical software systems built on such libraries. We also evaluated TAINTCRYPT on 5 popular tools and libraries, which generated new security insights. Our experimental evaluation on large code bases indicates the scalability of our approach.

2 MOTIVATION AND THREAT MODEL

To motivate this work, in this section, we present few examples of cryptographic vulnerabilities from real-world software. Then, we present our threat model.

2.1 Motivating Examples

Like other types of security vulnerabilities, one of the common causes of vulnerable information flows in cryptographic implementations is their inclusion of basic programming errors.

Example 1. For example, consider the code snippet excerpted from the core ScreenOS 6.2 PRNG functions [6] in Figure 1. In this case, the shared use of global variables (`prng_temporary` and `prng_output_index`) causes the leak of sensitive data `prng_seed` (Line 5 in `prng_reseed`) in the immediate post-seed (Line 16) output of function `prng_generate`. As another case of this kind, a memory disclosure vulnerability called *heartbleed* in OpenSSL (e.g., CVE-2014-0160) had the potential of leaking sensitive information (*e.g.*, cryptographic keys, and PRNG seeds). In fact, vulnerabilities and security threats rooted in similar coding errors are commonly found in real-world cryptographic software. Dealing with these issues can be particularly challenging as oftentimes addressing one problem can lead to other problems due to the bug fixes [24]. As an essential step towards overcoming such challenges, we will show how static information flow analysis can be employed to detect various types of sensitive data leakage in cryptographic code (Section 4).

Example 2. The first example as shown in Figure 1 illustrates the cases in which the vulnerable cryptographic information flows could be manually inspected. In large-scale projects that involve hundreds of developers, however, it is very difficult or impractical to check each of the information flow paths manually. For example, OpenSSL (as of commit $5748e4dc$) consists of $7,157$ functions, totaling $325,000$ lines of code (LoC). In addition, this software project involved 323 collaborating contributors. For sizable cryptographic software, manual approaches would not be feasible whereas automated security defense/enforcement mechanisms are mandatory.

```
1  void prng_reseed(void) {
2    ...
3    if (dualec_generate(prng_temporary, 32) != 32)
4      error_handler("ERROR:␣unable␣to␣reseed", 11);
5    memcpy(prng_seed, prng_temporary, 8);
6    prng_output_index = 8;
7    memcpy(prng_key, &prng_temporary[prng_output_index], 24);
8    prng_output_index = 32;
9  }
10
11 void prng_generate(int is_one_stage) {
12   ...
13   prng_output_index = 0;
14
15   if (!one_stage_rng(is_one_stage)) {
16     prng_reseed();
17   }
18
19   for (; prng_output_index <= 0x1F; prng_output_index += 8) {
20     // FIPS checks...
21     x9_31_generate_block(time, prng_seed, prng_key, prng_block);
22     // FIPS checks...
23     memcpy(&prng_temporary[prng_output_index], prng_block, 8);
24   }
25 }
```

Fig. 1: Excerpt from a real-world cryptographic program (the core ScreenOS 6.2 PRNG functions [6]), where `prng_temporary` and `prng_output_index` are global variables. When `prng_reseed` is called (Line 16), the loop variable `prng_output_index` in function `prng_generate` is set to 32, causing `prng_generate` to output sensitive data `prng_seed` at Line 5.

2.2 Threat Model

In this paper, we focus on two main categories of threats associated with cryptographic code that are relevant to vulnerable information flows. These threats thus could be addressed through properly designed static information flow analyses.

- **Library-level coding vulnerabilities.** Coding errors (such as premature exit of a for loop due to an incorrect loop condition [6]), vulnerabilities (such as key leakage), configuration issues and misuses (such as insecure storage of secrets) in third-party cryptographic libraries are particularly dangerous, as the code is usually widely used, oftentimes in commercial systems. As a consequence, security threats at this level can potentially put at risk a range of applications that are built on vulnerable libraries.
- **Application-level implementation vulnerabilities.** Application code varies (e.g., Android apps, client-side software, web applications), and there is a large variety in them. Vulnerabilities in this category are mostly related to API misuses—for example, erroneously invoking or configuring SSL library APIs [25], using obsolete crypto primitives, or intentionally disabling security mechanisms.

We assume that developers of both third-party cryptographic libraries and higher-level cryptographic applications could write vulnerable code. Within a single tool, we aim to cover as many vulnerabilities as possible, which can be used by developers who are clueless, as well as developers who are more experienced but may still write insecure code (e.g., PRNG seed leakage [1] or Heartbleed [5]). We do not target side-channel vulnerabilities (e.g., RSA padding oracle [26]) since our analysis is static in nature, while static analysis is inherently limited in dealing with those intrinsically dynamic issues [23]. However, we show that some of the straightforward cases (e.g., early termination or exposing the error conditions can be mapped within our general framework.

3 Crypto Vulnerabilities

In this section, we present different state-of-the-art cryptographic vulnerabilities. We also categorize them into several broader groups. Further, in Table 1, we present a set of security rules that ought to be enforced to defend crypto implementations against these vulnerabilities. The identification of the vulnerabilities are based on the exploratory study. Cryptographic vulnerabilities are included from the state-of-the-art cryptographic vulnerabilities. Programming errors are included by observing the change-logs of various OpenSSL's releases to find interesting cases. This types of exploratory study are not new in the literature [8], [9], [10], [11].

3.1 Chosen-plaintext attacks on IVs

Electronic Codebook (ECB) mode encryption is not semantically secure [27]. Bard *et al.* [28] showed that, the determinism of initialization vectors (IVs) can make cipher block chaining (CBC) mode encryption insecure too. However, the vulnerability remained merely hypothetical, until late 2011 when Doung and Rizzo [29] demonstrated a live attack (known as BEAST) against PayPal by exploiting the vulnerability. Row 1 of Table 1 corresponds to the security enforcement rule to avoid the use of ECB mode cipher and Row 2 corresponds to the attacks related to the predictability of IVs in CBC mode encryption. In Section 4, we present static information flow analysis based mechanisms to detect these vulnerabilities.

3.2 Attacks on PRNG

Historically, random number generators have been a major source of cryptographic information flow vulnerabilities [30], [31], [32]. The reason is that many of the cryptographic schemes rely on a cryptographically secure random number generator for the key and cryptographic nonce generation (Row 11 of Table 1). A random number generator can be exploited such that its behaviors are made predictable. When these attacks occur, such vulnerabilities as the use of predictable seeds (Row 9) and backdoor-able PRNG (Row 10) can be manipulated by an attacker as a backdoor to break the security of the cryptographic applications that use the randomly generated numbers resulted from the PRNG.

The NIST standard for PRNG referred to as *Dual EC PRNG*, has been considered both biased and backdoor-able by the security community [33]. Researchers have shown that the backdoor-ability of Dual EC PRNG was the main reason behind the Juniper incident in 2015 [6], and they also revealed how the cascade of multiple vulnerabilities due to programming errors led to the leak of PRNG seeds in Juniper Network (Row 19). We will demonstrate how the proposed static program analysis can be leveraged to detect such vulnerabilities (Section 4).

3.3 Use of Legacy Ciphers

There are several attacks based on the use of legacy ciphers, where cryptanalysis is feasible. For example, the Logjam attack [34] allows a man-in-the-middle attacker to downgrade vulnerable TLS connections to 512-bit *export-grade* cryptography. In [35], the

TABLE 1: Enforceable security rules in different cryptographic implementations. * indicates a rule focusing on data integrity and # indicates a rule focusing on data secrecy protection. Here, CPA and CCA stand for chosen plaintext attack and chosen ciphertext attack, respectively. (✓) indicates that the rule is implemented in TAINTCRYPT.

Attack Type	Enforceable Rule	Crypto property	Static Analysis Tool
CPA	(1) Should not use ECB mode in symmetric ciphers*	Secrecy	Taint Analysis (✓)
	(2) IVs in CBC mode, should be generated randomly*	Secrecy	Taint Analysis
CCA	(3) Validity of ciphertexts should not be revealed in symmetric ciphers	Secrecy	Program Dependence Analysis
	(4) Validity of ciphertexts should not be revealed in RSA	Authentication	Program Dependence Analysis
	(5) Should not use *export grade* or broken asymmetric ciphers*	Authentication	Data Flow Analysis
	(6) Should not use 64 bit block ciphers (e.g., DES, IDEA, Blowfish)*	Secrecy	Taint Analysis (✓)
	(7) Should not allow early termination (timing side channels)	Secrecy	Program Dependence Analysis
	(8) Should not allow cache-based side channels	Secrecy	–
Predictability	(9) PRNG seeds should not be predictable*	Randomness	Taint Analysis
	(10) Should not use untrusted PRNGs*	Randomness	Taint Analysis (✓)
	(11) Nonces should be generated randomly*	Randomness	Taint analysis (✓)
Memory Corruption	(12) Should not allow double "free()" exploit*	Determinism	Taint Analysis (✓)
	(13) Should not have type truncation (e.g., 64 bit to 32 bit integers)	Determinism	Data Flow Analysis
	(14) Should not leave any wild or dangling pointers	Determinism	Data Flow Analysis
	(15) Should guard against Integer overflow*	Determinism	Taint Analysis
	(16) Should not write to a memory (buffer) beyond its length*	Determinism	Taint Analysis
Crash	(17) Should Check return values of untrusted codes/libraries*	Availability	Taint Analysis
	(18) Division operations should not be exposed to arbitrary inputs*	Availability	Taint Analysis
Data Leak	(19) Should not leak sensitive data#	Secrecy	Taint Analysis (✓)
Key Leak	(20) Should not use predictable/constant cryptographic keys	Secrecy	Data Flow Analysis
Memory Leak	(21) Should not leave allocated memory without freeing	Availability	Data Flow Analysis
Memory Disclosure	(22) Should not read to a memory beyond its length (heartbleed)*	Secrecy	Taint Analysis (✓)
Hash Collision	(23) Should not use broken hash functions*	Integrity	Taint Analysis (✓)
Stack Overflow	(24) Cyclic function calls should not depend on untrusted inputs	Availability	Program Dependence Analysis
State machine Vulnerabilities	(25) Should detect illegal transitions in protocol state machines	Authentication	–

authors demonstrated the recovery of a secret session cookie by eavesdropping HTTPS connections. Prior research also demonstrated that the use of weak hash functions (e.g., MD5 or SHA-1 in TLS, IKE, and SSH) might cause almost-practical impersonation and downgrade attacks in TLS 1.1, IKEv2, and SSH-2 [36]. These attacks are characterized in Table 1: Rows 5 (asymmetric cipher), 6 (symmetric cipher), and 23 (hash functions). The corresponding vulnerabilities can be detected using our static analysis as described in Section 4.

3.4 Padding Oracles

Padding Oracle vulnerabilities can be categorized into two classes: (1) padding oracles in symmetric ciphers and (2) padding oracles in asymmetric ciphers.

Padding oracles in symmetric ciphers. Vaudenay *et al.* [37] presented a decryption oracle out of the receiver's reaction on a ciphertext in the case of valid/invalid padding of CBC mode encryption. In SSL/TLS protocol, the receiver may send a *decryption failure* alert, if invalid padding is encountered. By exploiting this information leaked from the server and cleverly changing the ciphertext, an attacker is able to decrypt a ciphertext without any knowledge of the key. "POODLE" [38] is a padding oracle attack that targets CBC-mode ciphers in SSLv3. "Lucky Thirteen" [39] is also a padding oracle attack on CBC-mode ciphers, exploiting the timing side channel vulnerabilities in victims that do not check the MAC for badly padded ciphertexts. In Row 3 of Table 1, we summarize padding oracle attacks on CBC mode encryptions.

Padding oracles in asymmetric ciphers. In [26], Bleichenbacher presented a stealthy attack on RSA based SSL cipher suites. The author utilized the strict structure of the PKCS#1 v1.5 format and showed that it is possible to decrypt the `PreMasterSecret` in a reasonable amount of time. There are numerous examples of using "Bleichenbacher padding oracle" to recover the RSA private key in different settings [40], [41], [42], [43], some of which use timing side channels to distinguish between properly-formed and malformed ciphertexts [44], [45].

In [21], authors proposed a data-flow analysis based technique to detect padding oracles due to non-constant-time implementations. In [22], authors proposed an efficient representation of Program Dependence Graph which can be utilized to verify constant-time implementations.

We characterize padding oracle attacks in (Rows 3 and 4) of Table 1. Although, results from [21] and [22] indicates that padding oracles from a class of non-constant-time implementation can be detected using program dependence graph analysis. However, verifying constant-time implementations to eliminate these side-channel exploitations is notoriously difficult, because of its indirect and complex dependency on program control flows [23].

3.5 Side-Channel Exploitations

We categorize side-channel attacks in cryptographic implementations into two classes: (1) timing-based (2) cache-based side-channel attacks.

Timing-based side-channel attacks. Brumley *et al.* [46] presented a timing based side channel attacks on OpenSSL's implementation of RSA decryption. In [47], the authors identified vulnerabilities to a timing attack in OpenSSL's ladder implementation for curves over binary fields. Exploiting these vulnerabilities, the authors demonstrated stealing the private key of a TLS server that authenticates with ECDSA signatures. Timing side-channels are hard to detect in general. However, some relatively straightforward cases, e.g., timing side channels due to early termination can be detected using program dependence analysis. Row 7 of Table 1 summarizes such timing-based side-channel attacks.

Cache-based Side-channel attacks. After the introduction of cache-based side-channels [48], researchers demonstrated the existence of side-channels in various cryptographic implementations (e.g., AES [49] and DSA [24]). In [24], the authors presented a cache-based side-channel to compromise the OpenSSL's implementation of the DSA signature scheme and recovered keys in TLS and SSH cryptographic protocols. Row 8 of Table 1 characterizes cache-based side-channel attacks in cryptographic implementations.

As discussed in Section 3.4, detecting side channels is an intrinsically difficult problem. TAINTCRYPT cannot detect timing side-channel vulnerabilities. TAINTCRYPT cannot detect these side-channel vulnerabilities.

3.6 State Machine Vulnerabilities

Attacks exist which exploit vulnerabilities in the protocol state machines of different cryptographic protocols [17], [18]. For example, the CCS injection attack [50] on OpenSSL's `ChangeCipherSpec` processing vulnerability allows malicious intermediate nodes to intercept encrypted data and decrypt them while forcing SSL clients to use weak keys that are exposed to the malicious nodes.

Different cipher-suits in TLS use different message sequences. In SKIP-TLS [18], TLS implementations incorrectly allow some messages to be skipped even though they are required for the selected cipher suite. The FREAK attack [51] has led to a server impersonation exploit against several mainstream browsers (including Safari and OpenSSL-based browsers on Android). Like most of the exploits of this category, FREAK also targets a class of deliberately chosen weak, export-grade cipher suites. These attacks are summarized in Row 25 of Table 1.

Most of the techniques [16], [17], [18] that detect vulnerabilities due to state machine exploitations use fuzzing-based input generation mechanisms based on *dynamic* program analyses. In contrast, building practical *static* analysis based detection mechanisms have yet to be investigated. A key challenge towards static detection lies in the fact that as the protocol's internal states increase, the computational complexity will accordingly rise exponentially.

3.7 Programming Errors

Programming errors have been a major source of vulnerabilities in C/C++ security software [52]. These vulnerabilities ranged from improper memory use to improper memory management. Examples of improper memory use include memory over-read (e.g., heartbleed attack [5]) (Row 22 of Table 1), memory over-write (e.g., buffer overflow [53], [54]) (Row 16), integer overflow [52] (Row 15), type truncation (Row 13)), and stack overflow[1] (Row 24). Example vulnerabilities that boil down to improper memory management are `malloc` without `free` (Row 21), double free [55] (Row 12), and dangling pointers (Row 14).

1. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0228

In addition, prior studies [8], [27] have shown that other programming errors, such as those that are due to careless handling of cryptographic keys (e.g., using hard-coded keys), are also prevalent in the wild (Row 25). In Section 4, we present how static program analysis can be used to detect cryptographic vulnerabilities that are induced by various programming errors.

4 SECURITY RULES AND ENFORCEMENT

In this section, we first present the enforceable security rules we derived against the various cryptographic code security vulnerabilities described in Section 3. Then we discuss how various types of static analysis techniques can be used to detect the violations of these rules. Once a technique is fixed, then we elaborate on how these rules can be expressed in a deterministic finite automaton (DFA) based language for enforcement. In particular, we demonstrate how security-aware testing can be enabled to enforce these rules via static code analysis.

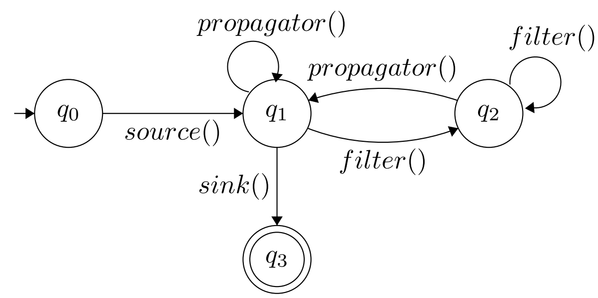


Fig. 2: Finite state machine (FSM) of taint analysis.

Enforceable security rules.

By analyzing different genres of attacks, we have identified 25 categories of cryptographic vulnerabilities and corresponding security rules that should be enforced in a cryptographic program to ensure different security properties, as shown in Table 1. These 25 categories of attacks fall in 12 higher-level attack classes (e.g., memory corruption and data leak) listed in the first column. Note that, the rules from memory corruption, crash, memory leak, and stack overflow are not cryptographic program specific, hence applicable for general program implementations. However, the violation of these rules in cryptography implementations causes violations in some of the cryptographic properties. To provide a one-stop service to secure cryptography implementations, we included these rules in our threat model.

4.1 Mapping Rules with Program Properties

23 out of the 25 security rules are enforceable through static code analysis. To use static analysis effectively for enforcing cryptographic rules we have to map cryptographic rules (meta-level properties) with static analysis properties. This mapping aims to capture sufficient condition of proving the violation of cryptographic rules. Since such a violation might not imply an exploitable vulnerability, thus it does not capture necessary conditions.

4.1.1 Detection with taint analysis

Next, we show that 15 out of these 23 rules can be enforced using static taint analysis, which requires mapping these rules to taint analysis properties. Specifically, we define the properties of taint analysis and map these rules with taint analysis properties.

TABLE 2: Transition functions (δ) of the finite state machine (FSM) presented in Figure 2.

		Inputs			
		source()	*propagator()*	*filter()*	*sink()*
States	q_0	q_1	–	–	–
	q_1	–	–	q_2	q_3
	q_2	–	q_1	–	–
	q_3	–	–	–	–

Taint analysis typically works by identifying *dangerous flows* of *untrusted* inputs into *sensitive* destinations [56]. Generally, a taint analyzer refers to four types of functions to identify these flows: sources, propagators, filters and sinks. A *source* is a function that produces an untrusted input, while a *sink* is a function that consumes an untrusted input sending it to a sensitive destination. A *propagator* is a function that propagates the untrusted data from one point of the program (via a variable) to another, while a *filter* is a function that purifies an *untrusted* variable and makes it *trustworthy*. Using taint analysis we can check the following two properties in a program.

- *Integrity.* The *integrity* property of a program regards to whether untrusted values (i.e., values generated from sources) can reach and modify trusted placeholders (i.e, sinks).
- *Confidentiality.* One may also be interested in the dual property of integrity such as *confidentiality* (i.e., whether values generated from sensitive sources can reach to untrusted sinks).

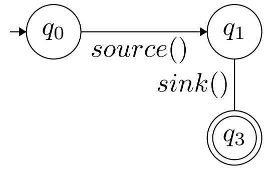
Finite state machines are among the natural choices to model cryptographic protocols. Hence, we choose to model the rules using finite state machines. Formally, we can define taint analysis as a deterministic finite automation (DFA) that can be represented by the tuple: $(Q, \sum, \delta, q_0, F)$. Where, $Q = \{q_0, q_1, q_2, q_3\}$, $\sum = \{source(), propagator(), filter(), sink()\}$, δ is presented in Table 2, $q_0 = \{q_0\}$ and $F = \{q_3\}$. Figure 2 shows the finite state machine (FSM) representation. In the next section, we discuss how this DFA based language can be used to express cryptographic properties at the meta-level so that taint analysis can be used to detect their violations.

Next, we discuss the mapping of cryptographic properties with taint analysis properties and show that taint analysis can be used effectively to detect any violations of these cryptographic properties.

Use of insecure primitives.

Generally, cryptographic libraries provide a high-level interface to support a wide range of cryptographic functions (e.g., hashing, symmetric ciphers, asymmetric ciphers, signature, etc.), so that the coding style remains consistent regardless of the underlying algorithm or mode. Most of them provide a set of convenient functions to create the specification of a crypto primitive (e.g, MD5) that can be used to initialize a certain type of cryptographic operation (e.g., digest).

To identify, the use of insecure cryptographic primitives (Rules 1, 6, 10, 23 in Table 1), one needs to identify that an insecure cryptographic primitive is used to initialize a cryptographic operation. If we define the creation of insecure cryptographic primitives as *source()*'s and the initialization of cryptographic operations as *sink()*'s, then the detection of such cases can be represented by a DFA tuple: $(Q, \sum, \delta, q_0, F)$. Where, $Q = \{q_0, q_1, q_3\}$, $\sum = \{source(), sink()\}$, δ is presented in Figure 3(b), $q_0 = \{q_0\}$ and $F = \{q_3\}$. The finite state machine is presented in Figure 3(a). Since, if we discard input symbol set {*propagator()*, *filter()*} from FSA of Figure 2, the FSM in Figure 2 and Figure 3(a) becomes equivalent. *This means that taint analysis can detect all such uses of insecure crypto primitives.*



(a)

		Inputs	
		$source()$	$sink()$
States	q_0	q_1	–
	q_1	–	q_3
	q_3	–	–

(b)

Fig. 3: (a) Finite state machine (FSM) and (b) transition function table (δ) to detect insecure cryptographic primitives. For example, to detect the usage of MD5, `EVP_md5` function can be used as a source and `EVP_DigestInit_ex` can be used as sink.

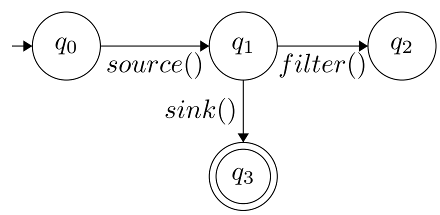
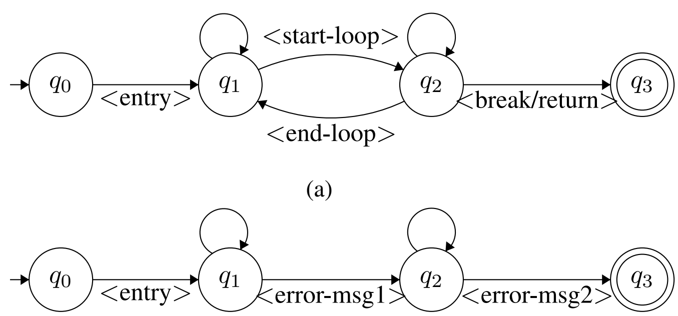


Fig. 4: The finite state machine (FSM) to detect unsanitized inputs from external sources. For an example, to defend against the heartbleed attacks in OpenSSL, the length parameter of `memcpy` should not be directly propagated (without filtering) from any untrusted source e.g., `n2s`, which is a function used to read values from the network.



(a)

(b)

Fig. 5: DFA to detect (a) early termination in a loop, (b) non-generic error messages by traversing program dependence graphs. Inputs of these DFA are the nodes of the graph. Here, <entry> represents the generic triggering node. For example, `EVP_DecryptFinal_ex` can serve as a trigger for rule 4. We labeled the inputs for only the state changing transitions.

Filtering data from external sources.

Data from external sources should be filtered before use (Rules 15, 16, 17, 18 and 22 of Table 1). If we define external sources as *source()* and functions that are sensitive to any external data as *sink()* and any sanitizing/filtering function as filters, then the FSM in Figure 4 can be used to detect any flow from external sources to the sensitive sink avoiding filters. Discarding input symbol *propagator()* from FSM of Figure 2 will result the FSM in Figure 4. Thus, taint analysis can detect all such violations.

Mappings for other rules can be deduced similarly.

4.1.2 Detection with other techniques.

In this section, we discuss some of the rules that can be can be detected using other static analysis techniques and expressed using deterministic finite automaton (DFA). The purpose of this is to motivate future research to unify various detection techniques within a single tool.

It is still unclear how the techniques proposed in [21], [22], [23] can be mapped within our general framework to detect timing-side channels. However, some of the straightforward cases can be mapped to our general framework. For example, an early termination using break/return or non-generic error message using program dependence graph analysis (Rule 3, 4 & 7), can be expressed in DFA based language as shown in Figure 5. It is unclear how to model early termination due to the loop control expressions (as shown below) using a DFA.

```
while(arr1[i] == arr2[i] && i < max_length){
  i++;
}
```

Static analysis can also be used to detect cyclic function calls on untrusted inputs (Rule 24). Type truncation (Rule 13), dangling pointers (Rule 14), memory leaks (Rule 21) can be detected using forward data flow analysis. The use of constant keys can be detected using backward data flow analysis (Rule 20) [11]. Once, the technique is fixed, then expressing these rules in DFA based language is relatively straightforward.

4.2 System Overview

In Section 4.1, maximum rules (15 out of 23) can be enforced using taint analysis. Hence, to demonstrate the effectiveness of our methodology, we built a static taint analysis based system (named, TAINTCRYPT) that can be used to automatically enforce all these 15 rules. TAINTCRYPT is built as a checker on top of the Clang static analyzer [57]. Clang static analyzer is a compile-time static analysis platform, which runs a set of checkers to find bugs during compilation of C/C++ programs [58]. TAINTCRYPT takes the cryptographic program under checking as input and outputs a security report that informs the detected cryptographic vulnerabilities in the program.

Specifically, TAINTCRYPT analyzes the input program in three key steps corresponding to the three technical components shown in the figure:

- Clang preprocessing, which transforms the given program written in C to its control flow graph (CFG).
- Symbolic execution, which explores the program symbolically and produces symbolic values for program states on the CFG. The execution is path-sensitive and every possible path through the program is explored. The explored execution traces are represented with `ExplodedGraph` object. Each node of the graph is `ExplodedNode`, which consists of a `ProgramPoint` and a `ProgramState`.
- Taint checking, which performs the static information flow analysis on `ExplodedGraph` of a given program to identify cryptographic vulnerabilities.

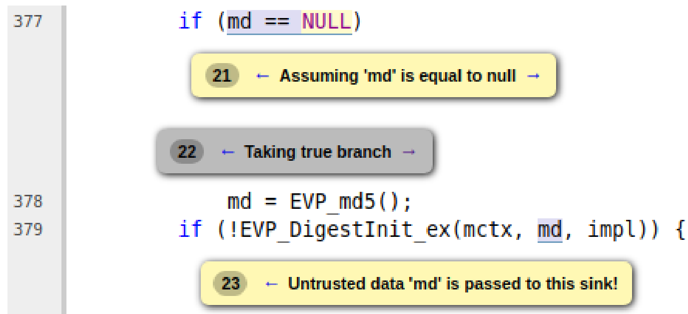


Fig. 6: An example of TAINTCRYPT detecting the use of vulnerable functionality (MD5) in OpenSSL, which violates the security rule against using broken hash functions (Row 23 of Table 1). In this example, our analysis correctly identified the violation by reporting the invocation of a vulnerable hash function `EVP_MD5()`.

To accommodate varied application scenarios, TAINTCRYPT reads a configuration file where users can specify taint *sources*, *sinks*, *propagators* and *filters* as functions used by the taint checking module. Note that, TAINTCRYPT has some built-in taint propagation rules. For example, (1) a variable will become tainted if a tainted value is assigned to it, (2) if the input of a built-in value transforming function (e.g., `atoi`, `atol`, `gets`, `toupper`, `tolower`) is tainted, then its return value is also marked as tainted, (3) if the input to a memory copying function (e.g., `memcpy`, `strcpy`) is tainted then its return value is also marked as tainted.

Note that, symbolic execution enables fine tracking of memory regions within clang static analyzer. In TAINTCRYPT an entire array is marked as tainted only if all the elements of the array are tainted, otherwise it marks individual elements at particular off-sets if the taint is propagated to the corresponding element. TAINTCRYPT leverages the builtin alias analysis of the clang static analyzer to compute aliases of a variable.

5 EVALUATION

We evaluate TAINTCRYPT by conducting a controlled experiment on known cryptographic vulnerabilities. Specifically, our evaluation answers the following questions.

- Can TAINTCRYPT detect known cryptographic vulnerabilities in popular libraries and tools? (Section 5.1)
- Can TAINTCRYPT detect new vulnerabilities from Cryptographic API misuses? (Section 5.2)
- In which scenarios can TAINTCRYPT produce false positives? (Section 5.3)

5.1 Controlled Experiments

The purpose of our evaluation is to demonstrate how TAINTCRYPT can be used effectively to enforce the 15 security rules that are enforceable through static taint analysis. In Table 3, we show the overview of our controlled experimental evaluation.

TABLE 3: Overview of TAINTCRYPT evaluation.

Property	Rule	Software	Version	# Violations	Similar Rules
Deprecated function invocation	(1) ECB mode	OpenSSL	1.0.1f	0	–
	(6) Insecure block ciphers	OpenSSL	1.0.1f	2	–
	(10) Insecure PRNG	OpenSSL	1.0.1f	0	–
	(23) Insecure Hash	OpenSSL	1.0.1f	7	–
Mandatory function invocation	(11) Random nonce	OpenSSL	1.0.1f	1	(2) Random IV (9) Non-predictable PRNG seed (20) Non-predictable keys
Untrustworthy inputs	(22) Memory disclosure	OpenSSL	1.0.1f	7	(15) Integer overflow (16) Buffer overflow (17) checking return values (18) Divide-by-zero
Unwanted call sequence	(12) Double `free()`	OpenSSL	1.1.0-stable	1	–
Sensitive data leak	(19) Leak of PRNG seeds	ScreenOS	6.2.0r1	1	–

```
n2s(p, payload);
    1  Within the expansion of the macro 'n2s': →
       a  Expression 'payload' gets tainted here
pl = p;

if (s->msg_callback)
    2  ← Taking false branch →
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
            &s->s3->rrec.data[0], s->s3->rrec.length,
            s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST)
    3  ← Assuming 'hbtype' is equal to 1 →
    4  ← Taking true branch →
        {
        unsigned char *buffer, *bp;
        int r;

        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
            5  ← Untrusted data 'payload' is passed to this sink!
```

Fig. 7: An example of TAINTCRYPT detecting the memory disclosure vulnerability in OpenSSL-1.0.1f (Row 22 of Table 1) hence the violation of the rule 22 against that vulnerability. Here the use of external data in variable `payload` without proper sanitization causes disclosure of memory of arbitrary size.

5.1.1 Use of Insecure Primitives

The enforcement of the security rules in Rows 1, 5, 6, 10 and 23 of Table 1 demands programmers to avoid/deprecate insecure cryptographic functionalities. For these cases, the user of TAINTCRYPT can specify the instantiation of insecure crypto primitives (e.g., `EVP_aes_128_ecb` (Rule 1), `EVP_rc4` (Rule 6), `EVP_md5` (Rule 23)) as sources and the initialization of any cryptographic operations (e.g., `EVP_EncryptInit_ex` (Rules

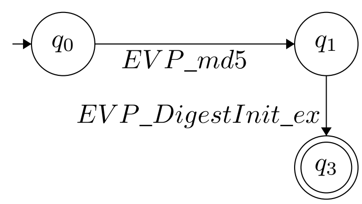


Fig. 8: Finite state machine (FSM) to detect the usage of MD5, `EVP_md5` function can be used as a source and `EVP_DigestInit_ex` can be used as sink.

1, 6) `EVP_DigestInit_ex`, `X509_digest` (Rule 23)) as sinks, and run the tool with this source/sink configuration. If there exists any information flow path between one of these listed sources and one of the specified sinks in the given code, TAINTCRYPT identifies and reports it. As discussed in Section 4.1.1, if TAINTCRYPT reports at least one such path, the requirement of deprecating the specified vulnerable functions is violated.

In Figure 8, we present the finite state machine (FSM) to detect the insecure usage of MD5 using TAINTCRYPT. Figure 6 presents an example of detected MD5 instance in OpenSSL[2]. For rule 10, TAINTCRYPT reports a violation if an use of `rand()` is observed.

5.1.2 Filtering Data From External Sources

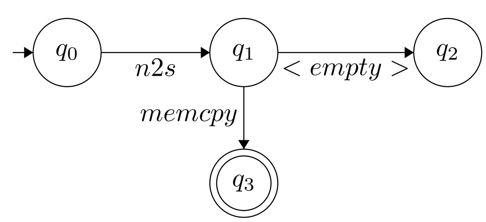


Fig. 9: The finite state machine (FSM) to detect heartbleed attack in OpenSSL. Here, `n2s` is a function used to read values from network.

Using data from external sources may be unavoidable, yet for security, such data should be filtered/sanitized before used. To enforce these types of security rules (in Rows 15, 16, 17, 18 and 22 of Table 1), the user of TAINTCRYPT would specify three

2. This vulnerability existed before commit $f8547f62$

types of functions in its configuration: (1) *untrusted* data sources (source functions), (2) their relevant sinks (sink functions) and, most importantly, (3) data filters/sanitizers (filter functions). If there exists any path from any source to any sink that bypasses all the filters, an instance of violation against the rules is reported by TAINTCRYPT.

As an example, Figure 7 illustrates the violation of rule 22 due to a heartbleed memory disclosure vulnerability in OpenSSL-1.0.1f. The source (function `n2s`) produces an untrusted input data in variable `payload` at Line 1464, which potentially reaches the invocation of the sink (function `memcpy`) at Line 1487 hence leads to memory disclosure. In this case, no filter is found on any potential flows from the source to the sink, thus TAINTCRYPT reports the rule 22 being violated. The corresponding FSM is shown in Figure 9.

```
call_dummy_source_client_random(s->s3->client_random);
   6  ← Expression 's->s3->client_random' gets tainted here →
p = s->s3->client_random;

/*
 * for DTLS if client_random is initialized, reuse it,
 * required to use same upon reply to HelloVerify
 */
if (SSL_IS_DTLS(s)) {
   7  ← Taking true branch →
```

(a) The source `call_dummy_source_client_random` produces untrusted input `s->s3->client_random`.

```

if (i && ssl_fill_hello_random(s, 0, p, sizeof(s->s3->client_random))
    return 0;
 *
 * For TLS 1.3 we always set the ClientHello version to 1.2 and rely 
 * supported_versions extension for the real supported versions.
 */
if (!WPACKET_put_bytes_u16(pkt, s->client_version)
        || !WPACKET_memcpy(pkt, s->s3->client_random, SSL3_RANDOM_SIZ
          11  ← Untrusted data 's->s3->client_random' is passed to this sink!
    SSLerr(SSL_F_TLS_CONSTRUCT_CLIENT_HELLO, ERR_R_INTERNAL_ERROR);
    return 0;
}
```

(b) `WPACKET_memcpy` is a potential sink and `ssl_fill_hello_random` a potential filter.

Fig. 10: An example of TAINTCRYPT facilitating the enforcement of security rule 11 which concerns secured random number generation in OpenSSL. In this case, as there exists a path from the source to the sink that avoids the filter `ssl_fill_hello_random`, TAINTCRYPT generates a warning reporting the rule being violated.

5.1.3 Ensuring Certain Function Invocations

To enforce the security rules 2, 9 and 11 (in corresponding rows of Table 1), we need to ensure the invocation of certain functions. To do that, the TAINTCRYPT user may specify these functions as *filters* as part of the configuration and then run the tool. This way, our analysis reports any *dangerous path* from a source to a sink that avoids these filters. If at least one such path is reported, the analysis indicates that the requirement of invoking the specified functions is violated.

To illustrate the ability of our analysis in detecting such violations, Figure 10 shows that the function `call_dummy_source_client_random` as a source produces an untrusted input held in the variable `s->s3->client_random`[3]. This untrusted data would flow to the sink `WPACKET_memcpy` if function `ssl_fill_hello_random` as a potential filter is not invoked. In this example, at least one path from the source to the sink exists which bypasses the filter. Since the existence of such a path is unexpected (with respect to the requirement that the filter should be invoked), TAINTCRYPT produces a warning indicating the security violation.

Closer inspection of this example reveals that the reported path actually reuses previously generated values of `s->s3->client_random` variable, thus cannot be considered as a security violation. This indicates that TAINTCRYPT might produce false positives when a violation of the defined cryptographic property does not directly translate to a security violation.

To *whitelist* such special value propagations, one should mark them as filter(s).

In our current prototype, only a sequence of 2 function calls can be modeled. However, TAINTCRYPT can be modified to record all the orthogonal call sequences on the way from source to sink, which can be used to analyze an arbitrary sequence of calls.

5.1.4 Preventing Data Leaks

Our analysis can also be used to detect violation of security rules (in particular, rule 19 in Table 1) against data leaks. To do so, the user would indicate in the configuration of TAINTCRYPT the sensitive data producers as *sensitive sources* and potential mole functions (e.g., function writing data to the filesystem or network) as *untrusted sinks*. The information flow analysis with respect to this configuration detects and reports if there exists any path from one of these sources to one of the sinks. In [6], the authors showed that ScreenOS of Juniper network was leaking seeds due to programming errors.

Figure 11 illustrates how our analysis with TAINTCRYPT can be employed to detect violations of rule 19. In this example, the data subject to potential leakage is a PRNG seed held in the variable `prng_seed`. The data is first tainted at Line 67 and later leaked at Line 98 via the sink `print_number` through five major steps highlighted in light yellow.

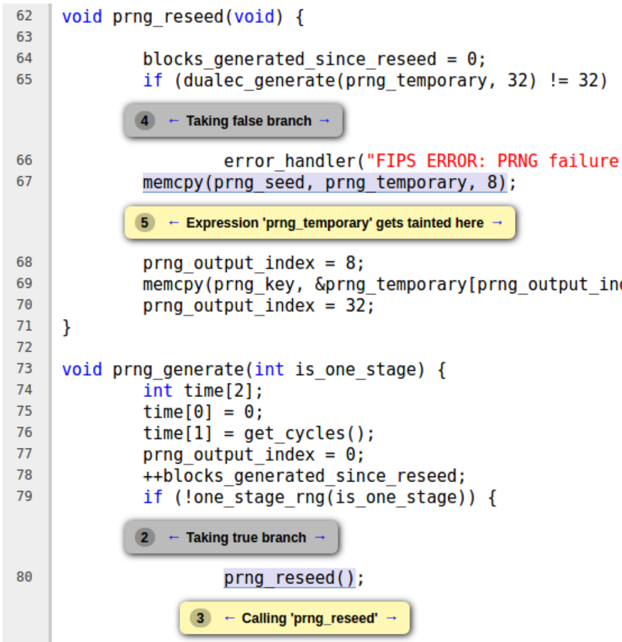
5.1.5 Avoiding Double-Free Vulnerabilities

Our technique can also accommodate the need for enforcing the security rule 12, which would prevent the vulnerabilities of double free in the given program. Specifically, the user of TAINTCRYPT may specify deallocation functions (e.g., the `free` function in C programs) as both sources and sinks, and then track the taint flow from the sources to the sinks. If a variable is passed as an argument to a deallocation function as a source, it gets *tainted*. Then if the taint propagates to a subsequent invocation of a deallocation function as a sink (i.e., the variable is freed the second time), an instance of violation of rule 12 is reported.

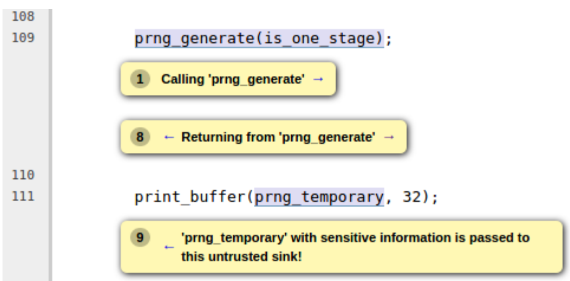
In Figure 12, the example illustrates the use scenario of TAINTCRYPT in which our analysis is applied to detect double-free vulnerabilities in OpenSSL[4]. In this example, the violation of rule 12 due to the variable `parms` being double-freed is detected

3. The present TAINTCRYPT prototype only accepts functions as sources. To accommodate the cases in which variables act immediately as taint sources, we use a dummy call that takes the variable as an argument to adapt to the current capabilities of TAINTCRYPT.

4. This vulnerability existed till commit *a34ac5b8*



(a) Sensitive source.



(b) Untrusted sink.

Fig. 11: An example illustrating the ability of our analysis in detecting and reporting an instance of data leak in Juniper Network. In this case, the sensitive data source in variable `prng_seed` as the first 8 bytes of variable `prng_temporary` reaches the sink `print_number`, which violates our security rule 19.

Rule	Violations	Software
(1) ECB mode	3	tor
(6) Insecure block ciphers	0	–
(10) Insecure PRNG	5	lib-apr
(23) Insecure Hash	2	httpd
(12) Double free()	0	–

TABLE 4: Vulnerabilities detected by TAINTCRYPT in 5 popular C/C++ projects (httpd, curl, lib-apr, openssh, tor).

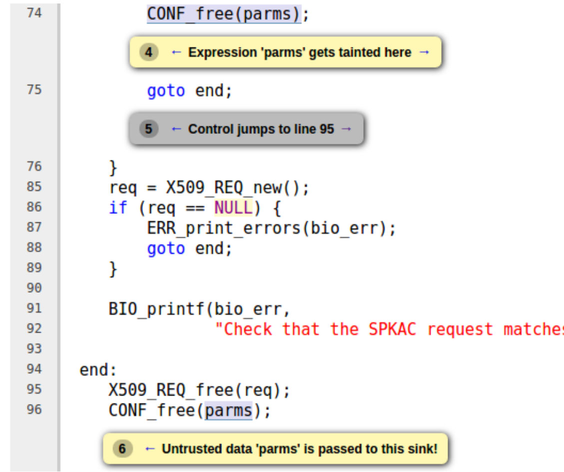


Fig. 12: An example case of our technique is used to detect a double-free vulnerability in OpenSSL, which constitutes an instance of violation of our security rule 12. Here TAINTCRYPT reports the double-free incident with variable `parms`.

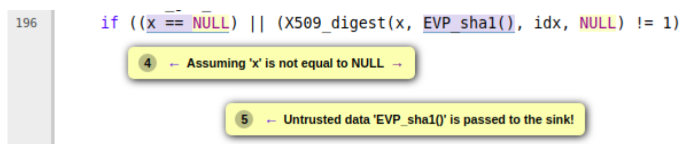


Fig. 13: An example of TAINTCRYPT detecting the use of SHA1 in httpd, which violates the security rule against using broken hash functions (Row 23 of Table 1).

through tracking information flow via this variable from the first deallocation site at Line 74 to the second at Line 96.

5.2 TAINTCRYPT in the wild

We ran TAINTCRYPT on 5 popular applications and libraries that are written in C/C++. These tools and libraries are httpd (version: 2.4.39), curl (version: 7.64.1), lib-apr (version: 1.7.0), openssh (version: 7.7p1), tor (version: 0.3.4.10). All these libraries and tools uses OpenSSL APIs for cryptographic functionalities. In this experiment we restrict TAINTCRYPT to find vulnerabilities under 5 rules (Rules 1, 6, 10, 12 and 23) presented in Table 4, since, only these 5 rules are based on APIs that are uniform across all the applications. The APIs are presented in Table 5.

TAINTCRYPT detected 2 usage of SHA1 (`EVP_sha1`) hash function under Rule 23 (Shown in Figure 13), 3 usage of AES in ECB mode (`EVP_aes_128_ecb`, `EVP_aes_192_ecb`, `EVP_aes_256_ecb`) under Rule 1. TAINTCRYPT also reported 5 usage of `rand()` under Rule 10. The summary of the analysis is presented in Table 4. Note that, we also manually analyzed the source code to determine false negatives (i.e., how many vulnerabilities did TAINTCRYPT miss). However, our manual analysis did not uncover any additional vulnerabilities.

Clang static analyzer is path-sensitive, it slows down the compilation process [57]. TAINTCRYPT inherits this impact on runtime overhead. However, our experimental evaluation on large scale projects indicates the scalability of this design choice.

5.3 Limitations

Because our analysis aims to capture sufficient conditions on meta-level properties, it has the potential to generate false positives. However, capturing necessary condition statically to prove cryptographic vulnerabilities is still open.

Also, static code analysis trades precision for soundness and scalability, in general. The symbolic execution based path-sensitive analysis takes computationally exponential time [59]. Therefore, considering the scalability, the loop unrolling mechanism of the SMT solver used to model symbolic execution in clang static analyzer is made constant bounded. Thus, similar to any other static analysis-based approaches, our technique suffers from imprecision as well.

Currently, our analysis only accepts functions as *sources*, *sinks*, *filters*, and *propagators*. As a result, in many real-world cryptographic software, the use of constraints as filters may be prevalent (e.g., using predicates to screen untrustworthy inputs) can lead to additional false positives. On the other hand, the comprehensiveness of these four lists of functions in the configuration for our technique immediately affect its soundness: missing some of these functions would lead to false negatives. For some of the rules, these configurations are standard across different code bases (e.g., Rules 1, 5, 6, 10, 12, 23), while for other rules developers need to specify these configurations.

Another limitation of TAINTCRYPT lies in its current implementation. TAINTCRYPT is built on the static taint checker in Clang, which by default does not support analysis across translational units. Thus, currently, our tool does not track taint propagation out of a translational unit. If a taint source and a taint sink are located in different translational units, TAINTCRYPT would not be able to detect the security violation when there is actually an information flow path from the source to the sink. However, this is an implementation flaw rather than a limitation of our technique itself.

6 RELATED WORK

Most existing approaches to defending against cryptographic vulnerabilities are based on dynamic analysis. A well-known approach is fuzzing (e.g., [14], [60]), a blackbox strategy which has been used for verifying hostname verification [14] and certificate validation in SSL/TLS [60]. Another notable example of dynamic approaches is to validate runtime protocol behaviors with a verifier, which is capable of detecting invalid or inconsistent network messages [15]. Although the work in [15] uses symbolic execution to infer client behaviors, it requires the concrete execution of programs for detecting anomalies.

Dynamic approaches rely on concrete executions of the program. Accordingly, their results are subject to the availability and quality of the run-time inputs that drive the executions, which may not be always available in practical use scenarios. In addition, when vulnerabilities are subtle and not externally visible, (i.e., do not manifest themselves in simple observable behaviors), dynamic solutions are ineffective. Examples of such cryptographic violations include the use of improper IVs in ciphers, poor random number generation, the leak of secrets, or the use of legacy cryptographic primitives in our threat model. Dynamic approaches are limited to finding only the input guided vulnerabilities with externally visible behaviors (e.g., triggering program crashes [16] or anomalous protocol states [15], [17]). In addition, as pointed out by [13], fuzzing and other dynamic analysis techniques typically cannot guarantee coverage, which may result in missed detection.

In [61], the authors explored the capability of symbolic execution to detect data authentication vulnerabilities in WPA2 implementations. Authors showed that mishandling data authentication may result in timing side channels (Rule 7) or decryption oracles (Rules 3, 4). Since our approach does not cover vulnerabilities related to decryption oracle or side channels, the work presented in [61] nicely compliments our tool, TAINTCRYPT.

In contrast, our approach to detect cryptographic security rule violations is purely static thus bypasses the above limitations of dynamic approaches. Moreover, static analysis has more potential to be scalable than dynamic analysis, as it does not require program execution (which always comes with extra overheads). Further, with extensive illustrations, we also have demonstrated the potential of static analysis to be capable of assisting developers with facilitating the enforcement of various security rules against corresponding cryptographic security vulnerabilities.

Prior works aiming to close the gap between the theory and practice of cryptography mostly target provable cryptographic solutions [7], mainly focuses on cryptographic API misuses [8], [62]. Egele *et al.* presented CryptoLint to detect cryptographic API misuses in Android applications [8] through lightweight control-flow analysis driven program slicing which has the potential to produce many false positives. Rahaman *et al.* proposed CryptoGuard [11] with an extended set of cryptographic misuse detection rules. CryptoGuard also proposes a set of refinement insights that leverages language restrictions and programming idioms to reduce false alarms. In [63], authors proposed a set of benchmarks to evaluate the performance of cryptographic misuse detection tools for Java/Android. FixDroid [9] and CogniCrypt [10] emphasize on the usability aspects of cryptographic misuse problems. While FixDroid focuses on improving the usability by facilitating real-time feedback and suggestion and CogniCrypt focuses on generating secure code. All these works focus on the misuse detection of cryptographic APIs in Java and Android by using program slicing. In contrast, TAINTCRYPT leverages taint-based information flow analysis to detect library-level and application-level vulnerabilities in C/C++ programs. The recent work SymCerts used a combination of concrete values and symbolic execution to detect missing checks in X.509 certificate verification code [13]. Concrete values are used to reduce the path exploration space.

Designing secure API wrappers has been shown to effectively eliminate the invocation of potentially vulnerable functions (e.g., unsafe memory copy) or operations (e.g., unsanitized SQL queries) at Google [64]. For Python, cryptography.io is a crypto library with simpler APIs, some of which require little to no configuration choices. These code refactoring approaches are useful for reducing misuses, however, they cannot address all the issues in our threat model, e.g., vulnerabilities in the library code or design flaws. In addition, user studies showed that simpler crypto APIs do not completely solve developers' problems [65].

Information flow analysis has been extensively used for detecting security vulnerabilities and threats, including both static (e.g., [66], [67]) and dynamic (e.g., [68], [69]) approaches. However, most of these existing techniques are developed for non-crypto related software problems, such as malware analysis [70], [71], [72] and vulnerability discoveries [73]. In addition, prior works applying information flow analysis are mostly limited to detecting sensitive and/or private data leaks. In comparison, we analyze a wide range of security vulnerabilities in cryptographic implementations and derive a number of enforceable security rules that immediately help developers prevent those vulnerabilities in their code. Additionally, our technical approach TAINTCRYPT can be utilized to detect violations against a variety of security rules including but not limited to those on data leaks.

7 CONCLUSION AND FUTURE WORK

A small programming error in the implementations can lead to dangerous security vulnerabilities that have a severe and broad impact on end-user devices and services. In this paper, we aimed to fill this gap by investigating real-world security threats in cryptographic code. Our result of this study was a categorization of 25 different types of cryptographic security vulnerabilities, along with associated defending rules that are practically enforceable. We showed that 23 out of 25 rules are enforceable using static analysis techniques. To facilitate developers in enforcing these rules in their cryptographic coding practice, we have further developed an information flow analysis technique TAINTCRYPT and implemented a prototype for C programs. We have demonstrated with a controlled evaluation of how our technique can be applied to varied use scenarios for identifying violations of 15 of our security rules and thus helping developers avoid corresponding vulnerabilities. Our experiment on 5 new tools and libraries using cryptographic APIs generated new security insights.

As future work, we plan to make TAINTCRYPT capable of detecting vulnerable cryptographic information flows across multiple translational units, with respect to the LLVM framework and Clang frontend on which our tool is built. In the longer term, we also plan to expand our technique to cover a larger and more diverse set of cryptographic vulnerabilities targeted by the remaining security rules that our current technique is not able to check. A promising approach toward that goal would be to leverage control and data flow analysis in cooperation with static tainting. Yet another part of future work is to improve the efficiency of TAINTCRYPT configuration by automatically discovering comprehensive lists of sources and sinks. Finally, supporting non-function sources and sinks would make our technique applicable in broader application scope.

8 ACKNOWLEDGMENT

This work was supported by the Office of Naval Research under Grant ONR-N00014-17-1-2498, NSF CNS 1657124.

REFERENCES

[1] S. Rahaman and D. Yao, "Program analysis of Cryptographic implementations for security," in *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*, 2017, pp. 61–68.

[2] "RFC 2818 - HTTP over TLS," https://tools.ietf.org/html/rfc2818, 2000, [Online; accessed 15-Oct-2018].

[3] "RFC 6944 - applicability statement: DNS security (DNSSEC) DNSKEY algorithm implementation status," https://tools.ietf.org/html/rfc6944, 2013, [Online; accessed 15-Oct-2018].

[4] "RFC 3207 - SMTP service extension for secure SMTP over transport layer security," https://www.ietf.org/rfc/rfc3207.txt, 2002, [Online; accessed 15-Oct-2018].

[5] "The Heartbleed Bug," *http://heartbleed.com/*, 2014, [Online; accessed 3-May-2017].

[6] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R. Weinmann, E. Rescorla, and H. Shacham, "A systematic analysis of the Juniper Dual EC incident," in *ACM CCS 2016*, 2016, pp. 468–479. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978395

[7] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering - Design Principles and Practical Applications*. Wiley, 2010. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470474246.html

[8] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *ACM CCS 2013*, 2013, pp. 73–84. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516693

[9] D. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, "A Stitch in Time: Supporting Android Developers in Writing Secure Code," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 1065–1077.

[10] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, "CogniCrypt: supporting developers in using cryptography," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 931–936.

[11] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, D. Yao, and M. Kantarcioglu, "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019 (to appear)*, 2019.

[12] 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 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 519–534.

[13] S. Y. Chau, O. Chowdhury, M. E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li, "SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations," in *IEEE S&P 2017*, 2017, pp. 503–520. [Online]. Available: https://doi.org/10.1109/SP.2017.40

[14] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "HVLearn: Automated Black-Box Analysis of Hostname Verification in SSL/TLS Implementations," in *IEEE S&P 2017*, 2017, pp. 521–538. [Online]. Available: https://doi.org/10.1109/SP.2017.46

[15] A. Chi, R. A. Cochran, M. Nesfield, M. K. Reiter, and C. Sturton, "A system to verify network behavior of known cryptographic clients," in *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, 2017, pp. 177–195. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/chi

[16] J. Somorovsky, "Systematic Fuzzing and Testing of TLS Libraries," in *ACM CCS 2016*, 2016, pp. 1492–1504. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978411

[17] J. de Ruiter and E. Poll, "Protocol State Fuzzing of TLS Implementations," in *USENIX Security 2015*, 2015, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[18] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 535–552.

[19] "Bleichenbacher's RSA signature forgery based on implementation error," https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html, 2006, [Online; accessed 15-Oct-2018].

[20] Y. Lindell and J. Katz, *Introduction to modern cryptography*. Chapman and Hall/CRC, 2014.

[21] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, 2014, pp. 1267–1279.

[22] B. Rodrigues, F. M. Q. Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, 2016, pp. 110–120.

[23] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying Constant-Time Implementations," in *USENIX Security 2016*, 2016, pp. 53–70. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[24] C. P. García, B. B. Brumley, and Y. Yarom, "Make Sure DSA Signing Exponentiations Really are Constant-Time," in *ACM CCS 2016*, 2016, pp. 1639–1650. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978420

[25] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "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*. ACM, 2012, pp. 38–49.

[26] D. Bleichenbacher, "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1," in *CRYPTO '98*, 1998, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1007/BFb0055716

[27] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why does cryptographic software fail?: a case study and open problems," in *APSys 2014*, 2014, pp. 7:1–7:7. [Online]. Available: http://doi.acm.org/10.1145/2637166.2637237

[28] G. V. Bard, "The Vulnerability of SSL to Chosen Plaintext Attack," *IACR Cryptology ePrint Archive*, vol. 2004, p. 111, 2004. [Online]. Available: http://eprint.iacr.org/2004/111

[29] "BEAST," https://vnhacker.blogspot.co.uk/2011/09/beast.html, 2011, [Online; accessed 3-May-2017].

[30] I. Goldberg and D. Wagner, "Randomness and the Netscape browser," *Dr Dobb's Journal-Software Tools for the Professional Programmer*, vol. 21, no. 1, pp. 66–71, 1996.

[31] D. J. Bernstein, Y. Chang, C. Cheng, L. Chou, N. Heninger, T. Lange, and N. van Someren, "Factoring RSA Keys from Certified Smart Cards: Coppersmith in the wild," in *ASIACRYPT 2013*, 2013, pp. 341–360. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-42045-0_18

[32] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," in *USENIX Security 2012*, 2012, pp. 205–220. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger

[33] S. Checkoway, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, H. Shacham, and M. Fredrikson, "On the practical exploitability of dual EC in TLS implementations," in *USENIX Security 2014*, 2014, pp. 319–335. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/checkoway

[34] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Z. Béguelin, and P. Zimmermann, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," in *ACM CCS 2015*, 2015, pp. 5–17. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813707

[35] K. Bhargavan and G. Leurent, "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN," in *ACM CCS 2016*, 2016, pp. 456–467. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978423

[36] ——, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH," in *NDSS 2016*, 2016. [Online]. Available: https://www.internetsociety.org/sites/default/files/blogs-media/transcript-collision-attacks-breaking-authentication-tls-ike-ssh.pdf

[37] S. Vaudenay, "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." in *EUROCRYPT 2002*, 2002, pp. 534–546. [Online]. Available: http://dx.doi.org/10.1007/3-540-46035-7_35

[38] B. Möller, T. Duong, and K. Kotowicz, "This POODLE bites: exploiting the SSL 3.0 fallback," 2014.

[39] N. J. AlFardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," in *IEEE S&P 2013*, 2013, pp. 526–540. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.42

[40] V. Klíma, O. Pokorný, and T. Rosa, "Attacking RSA-based Sessions in SSL/TLS," in *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, 2003, pp. 426–440. [Online]. Available: https://doi.org/10.1007/978-3-540-45238-6_33

[41] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J. Tsay, "Efficient Padding Oracle Attacks on Cryptographic Hardware," in *CRYPTO 2012*, 2012, pp. 608–625. [Online]. Available: https://doi.org/10.1007/978-3-642-32009-5_36

[42] T. Jager, S. Schinzel, and J. Somorovsky, "Bleichenbacher's Attack Strikes again: Breaking PKCS#1 v1.5 in XML Encryption," in *ESORICS 2012*, 2012, pp. 752–769. [Online]. Available: https://doi.org/10.1007/978-3-642-33167-1_43

[43] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt, "DROWN: breaking TLS Using SSLv2," in *USENIX Security 2016*, 2016, pp. 689–706. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aviram

[44] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews, "Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks," in *USENIX Security 2014*, 2014, pp. 733–748. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer

[45] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-Tenant Side-Channel Attacks in PaaS Clouds," in *ACM CCS 2014*, 2014, pp. 990–1003. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660356

[46] D. Brumley and D. Boneh, "Remote Timing Attacks Are Practical," in *USENIX Security 2003*, 2003. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical

[47] B. B. Brumley and N. Tuveri, "Remote Timing Attacks Are Still Practical," in *ESORICS 2011*, 2011, pp. 355–371. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23822-2_20

[48] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," *IACR Cryptology ePrint Archive*, vol. 2002, p. 169, 2002. [Online]. Available: http://eprint.iacr.org/2002/169

[49] D. J. Bernstein, "Cache-timing attacks on AES," http://cr.yp.to/papers.html#cachetiming, 2005, [Online; accessed 4-May-2017].

[50] "CCS injection vulnerability," *http://ccsinjection.lepidum.co.jp/*, 2015, [Online; accessed 4-May-2017].

[51] "The FREAK attack," *https://censys.io/blog/freak*, 2015, [Online; accessed 4-May-2017].

[52] S. Chen, J. Xu, and E. C. Sezer, "Non-Control-Data Attacks Are Realistic Threats," in *USENIX Security 2005*, 2005.

[53] "libevent (stack) buffer overflow in evutil_parse_sockaddr_port()," https://github.com/libevent/libevent/issues/318, 2016, [Online; accessed 4-May-2017].

[54] "Fixed potential stack corruption in `mbedtls_x509write_crt_der()`," https://github.com/ARMmbed/mbedtls/blob/development/ChangeLog#L118, 2016, [Online; accessed 4-May-2017].

[55] "Fixed pthread implementation to avoid unintended double initialisations and double frees." https://github.com/ARMmbed/mbedtls/blob/development/ChangeLog#L154, 2016, [Online; accessed 4-May-2017].

[56] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," in *NDSS 2007*, 2007. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/07/papers/cross-site-scripting_prevention.pdf

[57] "Clang static analyzer," https://clang-analyzer.llvm.org/, [Online; accessed 5-July-2019].

[58] M. Arroyo, F. Chiotta, and F. Bavera, "An user configurable clang static analyzer taint checker," in *35th International Conference of the Chilean Computer Science Society, SCCC 2016, Valparaíso, Chile, October 10-14, 2016*, 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1109/SCCC.2016.7835996

[59] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *USENIX OSDI 2008*, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[60] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "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*, 2012, pp. 38–49.

[61] M. Vanhoef and F. Piessens, "Symbolic Execution of Security Protocol Implementations: Handling Cryptographic Primitives," in *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018.*, 2018.

[62] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, ""Jumping Through Hoops": Why do Java Developers Struggle with Cryptography APIs?" in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 935–946.

[63] S. Afrose, S. Rahaman, and D. Yao, "CryptoAPI-Bench: A comprehensive benchmark on Java Cryptographic API misuses," in *IEEE Cybersecurity Development, SecDev 2019 (to appear)*, 2019.

[64] C. Kern, "Secure design: A better bug repellent," in *Proceedings of the IEEE Secure Development Conference (SecDev)*, 2017, keynote.

[65] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic apis," in *Proceedings of the 38th IEEE Symposium on Security and Privacy*, 2017.

[66] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 87–97.

[67] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information Flow Analysis of Android applications in DroidSafe." in *NDSS*, 2015.

[68] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007, pp. 196–206.

[69] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[70] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang, "Profiling user-trigger dependence for Android malware detection," *Computers & Security*, vol. 49, pp. 255–273, 2015. [Online]. Available: https://doi.org/10.1016/j.cose.2014.11.001

[71] K. Xu, D. D. Yao, B. G. Ryder, and K. Tian, "Probabilistic Program Modeling for High-Precision Anomaly Classification," in *IEEE CSF 2015*, 2015, pp. 497–511. [Online]. Available: https://doi.org/10.1109/CSF.2015.37

[72] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2017.

[73] Y. Kwon, B. Saltaformaggio, I. L. Kim, K. H. Lee, X. Zhang, and D. Xu, "A2c: Self destructing exploit executions via input perturbation," in *NDSS 2017*, 2017.

APPENDIX

Rule	Source(s)	Sink(s)
(1) ECB mode	EVP_aes_128_ecb, EVP_aes_256_ecb, EVP_aes_192_ecb	EVP_EncryptInit_ex
(6) Insecure block ciphers	EVP_des_cbc, EVP_des_ede_cbc, EVP_des_ede3_cbc, EVP_des_ecb, EVP_des_ede, EVP_des_ede3, EVP_rc2_cbc, EVP_rc2_ecb, EVP_rc2_64_cbc, EVP_rc2_40_cbc, EVP_rc4, EVP_rc4_40	EVP_EncryptInit_ex
(10) Insecure PRNG	rand	*
(23) Insecure Hash	EVP_md5, EVP_sha1	EVP_DigestInit_ex, EVP_DigestInit_ex
(12) Double free()	free, CONF_free	free, CONF_free

TABLE 5: APIs used as sources and sinks and their corresponding rules. (*) indicates any.