# Program Analysis of Cryptographic Implementations for Security

Sazzadur Rahaman
*Department of Computer Science*
*Virginia Tech*
sazzad14@cs.vt.edu

Danfeng (Daphne) Yao
*Department of Computer Science*
*Virginia Tech*
danfeng@cs.vt.edu

***Abstract*—Cryptographic implementation errors in popular open source libraries (e.g., OpenSSL, GnuTLS, BotanTLS, etc.) and the misuses of cryptographic primitives (e.g., as in Juniper Network) have been the major source of vulnerabilities in the wild. These serious problems prompt the need for new compile-time security checking. Such security enforcements demand the study of various cryptographic properties and their mapping into enforceable program analysis rules. We refer to this new security approach as *cryptographic program analysis (CPA)*. In this paper, we show how cryptographic program analysis can be performed effectively and its security applications. Specifically, we systematically investigate different threat categories on various cryptographic implementations and their usages. Then, we derive various security rules, which are enforceable by program analysis tools during code compilation. We also demonstrate the capabilities of static taint analysis to enforce most of these security rules and provide a prototype implementation. We point out promising future research and development directions in this new area of cryptographic program analysis.**

## 1. Introduction

In contrast to the multi-decade advancement of modern cryptography, the practical task of securing cryptographic implementation is still in its infancy. This gap became particularly alarming, after multiple recent high-profile discoveries of cryptography-related vulnerabilities in widely used network libraries and tools (e.g., heartbleed vulnerability in OpenSSL [1], seed leaking in Juniper Network [2]).

Surprisingly, most of these vulnerabilities occur due to programming mistakes rather than their underlying crypto algorithms [3]. For example, during the investigation of Juniper incident in 2015 (CVE-2015-7756) [2], researchers discovered that, Juniper Network code did not only contain backdoor-able cryptographic primitives, but also a fatal programming error that could leak PRNG seeds. In Listing 1, we see that the logical error incorporated due to the shared use of global variables (*e.g.*,`prng_temporary` and `prng_output_index`) causes the leak of `prng_seed` (Line 5 in `prng_reseed`) in immediate post-seed (Line 19) output of `prng_generate` function. In OpenSSL, a memory disclosure vulnerability named heartbleed (CVE-2014-0160) had the potential to leak sensitive information (*e.g.*, cryptographic keys, PRNG seeds, etc.). One can observe that similar types of bugs (or vulnerabilities) appear from time to time. Sometimes fixing one problem leads to another [4]. In general, these vulnerabilities due to simple programming errors in cryptographic implementations affect millions of devices, rendering millions of users vulnerable to adversarial attacks.

Listing 1: The core ScreenOS 6.2 PRNG functions [2]. Here, `prng_temporary` and `prng_output_index` are global variables. When `prng_reseed` is called (Line 19), the loop control variable (`prng_output_index`) of function, `prng_generate` is set to 32, causing `prng_generate` to output `prng_seed` from Line 5. In Section 3, We show how static taint analysis can be employed to detect these types of sensitive data leakage.

```
1  void prng_reseed(void) {
2    blocks_generated_since_reseed = 0;
3    if (dualec_generate(prng_temporary, 32) != 32)
4      error_handler("FIPS ERROR: PRNG failure, 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   int time[2];
13   time[0] = 0;
14   time[1] = get_cycles();
15   prng_output_index = 0;
16   ++blocks_generated_since_reseed;
17
18   if (!one_stage_rng(is_one_stage)) {
19     prng_reseed();
20   }
21
22   for (; prng_output_index <= 0x1F; prng_output_index += 8) {
23     // FIPS checks...
24     x9_31_generate_block(time, prng_seed, prng_key, prng_block);
25     // FIPS checks...
26     memcpy(&prng_temporary[prng_output_index], prng_block, 8);
27   }
28 }
```

With the increase of computational power, fuzzing and symbolic execution based approaches are being popular to automatically discover vulnerabilities [5], [6]. In [7], researchers presented a mechanism to infer client behaviors by leveraging symbolic executions of client-side codes. The work also demonstrated how such prediction can be used to filter malicious traffics to the server. However, these techniques are limited to find only the input guided vulnerabilities with externally visible behaviors (e.g., triggering program crashes [8] or anomalous protocol states [7], [9]). It is still unclear how to use them to enforce security rules in a code base, where the violations are not externally visible. For example, guarding against buffer over-read (Heartbleed), use of "improper" IVs in ciphers, "poor" random number generation or use of legacy cryptographic

primitives, etc.

Although effort to close the gap between the theory and practice of cryptography has been recognized previously [10], most of the work was on using and deploying provable cryptographic solutions in emerging applications, not on ensuring the correct implementation/use [11]. In [12], researchers presented an empirical study on Android applications analyzing some cryptographic misuses using lightweight control flow analysis. However, its capability is limited to detecting hard-coded seeds, keys, and IVs, the use of AES in ECB mode and using too few iterations for password-based encryption.

Static code analysis has been a central focus in scientific studies regarding malware analysis [13], [14], vulnerability discoveries [15], [16], [17], data leak detections [18], etc. It also has the potential to check and report whether a piece of code complies with certain rules or not [19], [20]. In this paper, we describe our initial efforts towards statically screening source code (namely C/C++ code) using static taint analysis to ensure the correct implementation of cryptographic properties (*i.e.*., *secrecy*, *pseudo-randomness*, etc.). We refer to this new security approach as *cryptographic program analysis (CPA)*.

Although appearing intuitive, this effort requires the challenging task of mapping abstract cryptography concepts to concrete program properties. We illustrate how to close this semantic gap by translating cryptographic concepts into properties those can be verified by taint-based program analysis tools. First, we study various state-of-the-art vulnerabilities to derive different security rules those are necessary to be complied by any cryptographic libraries, protocol implementations and any code that uses these cryptographic implementations. Then, we discuss static taint analysis-based methodology to enforce these security rules. We also provide a prototype implementation and demonstrate the effectiveness of this approach. Unlike [12], our static analysis tool is capable of path sensitive and context sensitive analysis, hence capable of enforcing a rich set of cryptographic properties.

Our technical contributions are summarized as follows.

- We provide a taxonomy of 25 types of exploitable vulnerabilities on cryptographic implementations. We categorize these vulnerabilities under 12 types of attacks. We derived 25 enforceable rules from this vulnerability analysis to defend 6 types of security properties. We discuss how static analysis can be useful to enforce these security properties. Our analysis shows that 15 out of these 25 rules are enforceable using static taint analysis.
- We design an LLVM-based toolset to facilitate static taint analysis, that can be used to enable security-aware testing to enforce various security rules in C/C++ based cryptographic implementations. As a case study, we show that how such security-aware testing can be realized in large-scale libraries like OpenSSL. We also discuss what problems remain unsolved in the light of our analysis and discuss future directions.

In order to perform complete automatic program analyses on cryptographic libraries and their usages in applications with millions of lines of code, more development work is still ahead. Our work reported in this paper addresses the fundamental challenge of mapping theoretical cryptographic concepts to practical code structures, which will guide future development effort of the community.

Our paper is organized as follows. In Section 2, we present different genres of vulnerabilities appeared in various cryptographic implementations. In Section 3, we present how different security rules can be enforced using static program analysis. We point out several promising future research and development directions in Section 4.

## 2. Crypto Vulnerabilities

In this section, we present different state-of-the-art cryptographic vulnerabilities. We also categorize them into several broad groups. In Table 1, we present a set of security rules those are needed to be enforced to defend crypto implementations against these vulnerabilities.

### 2.1. Chosen-plaintext attacks on IVs

Electronic Codebook (ECB) mode encryption is not semantically secured [3]. Bard *et al.* [21] showed that, the determinism of IVs can make cipher block chaining (CBC) mode encryption unsecured too. However, the vulnerability was theoretical until late 2011. Doung and Rizzo [22] demonstrated a live attack (known as BEAST) against Paypal by exploiting the vulnerability. Row 1 in 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 3, we discuss static taint analysis based mechanisms to detect these vulnerabilities.

### 2.2. Attacks on PRNG

Historically, random number generators have been a major source of vulnerabilities [23], [24], [25]. This is because many of the cryptographic schemes depend on a cryptographically secure random number generator for key and cryptographic nonce generation (Row 11), if a random number generator can be made predictable (e.g., use of predictable seeds (Row 9), backdoor-able PRNG (Row 10)), it can be used as a backdoor by an attacker to break the security.

The NIST standard for pseudo-random number generation named "Dual EC PRNG" has been identified by the security community as biased and backdoor-able [26]. In [2] authors showed that this backdoor-ability of *Dual EC PRNG* was the main reason behind the Juniper incident in 2015. In [2], authors also showed that how the cascade of multiple vulnerabilities due to programming errors lead to the leak of seeds in Juniper Network (Row 19). In Section 3, we show how static taint analysis can be utilized to detect such vulnerabilities.

### 2.3. Using Legacy Ciphers

There are several attacks based on the use of legacy ciphers, where cryptanalysis is feasible. For example, the

TABLE 1: Enforceable security rules in different cryptographic implementations. (*) indicates data integrity and (#) indicates data secrecy protection.

| Attack Type | Serial No | 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 | Unknown |
| | 4 | Validity of ciphertexts should not be revealed in RSA | Authentication | Unknown |
| | 5 | Should not use `export grade` or broken asymmetric ciphers* | Authentication | Taint Analysis |
| | 6 | Should not use 64 bit block ciphers (e.g., DES, IDEA, Blowfish)* | Secrecy | Taint Analysis |
| | 7 | Should not have timing side channels | Secrecy | Unknown |
| | 8 | Should not allow cache-based side channels | Secrecy | Unknown |
| 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 |
| Mem. Corrup. | 12 | Should not allow double "free()" exploit* | Deterministic behavior | Taint Analysis |
| | 13 | Should not have type truncations (e.g., 64 bit to 32 bit integers) | Deterministic behavior | Data Flow Analysis |
| | 14 | Should not leave any wild or dangling pointers | Deterministic behavior | Data Flow Analysis |
| | 15 | Should guard against Integer overflow* | Deterministic behavior | Taint Analysis |
| | 16 | Should not write to a memory (buffer) beyond its length* | Deterministic behavior | 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 |
| Mem. Leak | 21 | Should not leave allocated memory without freeing | Availability | Data Flow Analysis |
| Mem. 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 | Should not have cyclic function calls | Availability | Call Graph Analysis |
| State machine Vulnerabilities | 25 | Should detect illegal transitions in protocol state machines | Authentication | Unknown |

Logjam attack [27] allows a man-in-the-middle attacker to downgrade vulnerable TLS connections to 512-bit *export-grade* cryptography. In [28] authors demonstrated the recovery of a secret session cookie by eavesdropping HTTPS connections. In [29] authors demonstrated that the use of weak hash functions such as 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. Row 5 (asymmetric cipher), 6 (symmetric cipher) and 23 (hash functions) corresponds to such attacks in Table 1. In Section 3, we present static taint analysis based approach to detect these vulnerabilities.

## 2.4. Padding Oracles

Padding Oracle vulnerabilities can be divided into two groups: (1) padding oracles in symmetric ciphers (2) padding oracles in asymmetric ciphers.

*Padding oracles in symmetric ciphers.* In [30], Vaudenay *et al.* 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 using this information leaked from the server and cleverly changing the ciphertext, an attacker is able to decrypt a ciphertext without the knowledge of the key. "POODLE" [31] is a padding oracle attack that targets CBC-mode ciphers in SSLv3. "Lucky Thirteen" [32] is also a padding oracle attack on CBC-mode ciphers, exploiting the timing side channel due to not checking 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 [33], Bleichenbacher presented a stealthy attack on RSA based SSL cipher suites. Bleichenbacher 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 [34], [35], [36], [37], where some of them uses timing side channels to distinguish between properly and formed ciphertexts [38], [39]. Row 4 corresponds to these Bleichenbacher padding oracle attacks in Table 1.

We believe that padding oracles due to obvious padding errors might be detected using static program analysis (e.g., data flow analysis), while it is not clear that how static program analysis can be feasible to detect more sophisticated padding oracles (e.g., timing-based side-channels).

### 2.5. Side-channel Exploitations

The presence of side-channel attacks in cryptographic implementations can be categorized as: (1) Timing-based (2) Cache-based side-channel attacks.

*Timing-based Side-channel attacks.* In [40], Brumley *et al.* presents a timing based side channel attacks on OpenSSL's implementation of RSA decryption. In [41], authors presented a timing attack vulnerability in OpenSSL's ladder implementation for curves over binary fields. Using the vulnerability, authors demonstrated stealing the private key of a TLS server where the server authenticates with ECDSA signatures. Row 7 corresponds to these timing-based side-channel attacks in Table 1.

*Cache-based Side-channel attacks.* After the introduction of cache-based side-channels [42], researchers demonstrated the existence of side-channels in various cryptographic implementations (e.g., AES [43], DSA [4]). In [4], authors presented a cache-based side-channel to compromise the OpenSSL's implementation of DSA signature scheme and recover keys in TLS and SSH cryptographic protocols. Row 8 in Table 1 corresponds to cache-based side-channel attacks in cryptographic implementations.

Unfortunately, verifying constant-time implementations to eliminate these side-channel exploitations is notoriously difficult, because of its indirect/complex dependency on program control flows [44]. As a result, building feasible static program analysis based techniques to verify constant-time implementation is still open.

### 2.6. State Machine Vulnerabilities

There are several attacks on exploiting the vulnerabilities in the protocol state machines of different cryptographic protocols [9], [45]. For example, CCS injection attack [46] 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 which are exposed to the malicious nodes. Different cipher-suits in TLS use different message sequences. In SKIP-TLS [45], TLS implementations incorrectly allow some messages to be skipped even though they are required for the selected cipher suite. FREAK [47] leads to a server impersonation exploits against several mainstream browsers (including Safari and OpenSSL-based browsers on Android). Like most of the exploits from this category, FREAK also targets a class of deliberately chosen weak, export-grade cipher suites. Row 25 in Table 1 corresponds to such attacks.

Most of the techniques [8], [9], [45] to detect vulnerabilities due to state machine bugs use fuzzing-based input generation mechanism to perform dynamic analyses. However, building feasible static analysis based detection mechanism is still open, because with the increase of the protocol internal states the computational complexity will rise exponentially.

### 2.7. Programming Errors

Programming errors in C/C++ environment has been a major source of vulnerabilities in security software [48]. These vulnerabilities are ranged from improper memory usages like, memory over-read (heartbleed attack [1]) (Row 22), memory over-write (buffer overflow [49] [50] (Row 16), integer overflow [48] (Row 15), type truncation (Row 13)), stack overflow[1] (Row 24) to improper memory managements like, `malloc` without `free` (Row 21), double free [51] (Row 12), dangling pointers (Row 14), etc.

Studies [3], [12] show that, other programming errors due to careless handling cryptographic keys (i.e., hard coding keys in the source) are also prevalent in the wild (Row 25). In Section 3, we present how static program analysis can be used to detect such vulnerabilities.

## 3. Static Taint Analysis-based Enforcement

In this section, we present various security rules those are needed to be enforced in a code base to defend against the vulnerabilities presented in Section 2. We discuss how static taint analysis can be used to statically enforce most of these security rules and present a system prototype named *TaintCrypt*. Using OpenSSL as a case study, we also demonstrate how static code analysis based security enforcement mechanism can effectively help us achieve various security goals.

### 3.1. Enforceable Security Rules.

By analyzing different genres of attacks, we have identified 25 categories of vulnerabilities and inferred corresponding security rules, those should be enforced in a cryptographic implementation to ensure different security properties. In Table 1, we present these security rules. We identified that among them, 20 of the security rules are possible to be enforced by static code analysis. Close inspection shows that 14 of these rules (marked as (*) in Table 1) advocates the protection of data "integrity" to prevent untrusted values (produced by untrusted/vulnerable functions) to reach certain program points and another rule (marked as (#) in Table 1) advocates the prevention of sensitive data propagation to the untrusted functions. In this paper, we show that static taint analysis can be effectively applied to model these 15 rules. In particular, we show how security-aware testing can be enabled to enforce these 15 rules using static taint analysis.

### 3.2. System Overview

We build a static taint analysis engine named *TaintCrypt*. *TaintCrypt* is built on top of Clang Static Analyzer [52]. In *TaintCrypt*, one can specify taint *sources*, *sinks*, *propagators* and *filters* as functions to run static taint analysis on a code base. Clang Static Analyzer internally

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

uses symbolic execution to perform path sensitive exploration to detect vulnerabilities. It employs simple tracking based Satisfiability Modulo Theories (SMT) solver to model symbolic execution.

```
390    RAND_add(data, i, 0);   /* put in the RSA key. */
391    OPENSSL_assert(enc->iv_len <= (int)sizeof(iv));
392    if (RAND_bytes(iv, enc->iv_len) <= 0) /* Generate a salt */
       13 ← Taking false branch →
393        goto err;
394    /*
395     * The 'iv' is used as the iv and as a salt.  It is NOT taken f
396     * the BytesToKey function
397     */
398    if (!EVP_BytesToKey(enc, EVP_md5(), iv, kstr, klen, 1, key, NUL
       14 ← Untrusted data 'EVP_md5()' is passed to this sink!
```

Figure 1: Detection of using vulnerable functionality (MD5) in OpenSSL (Row 23). Our taint analysis engine reports the use of the vulnerable `EVP_MD5()` function.

Taint analysis is typically used to identify *dangerous flows* of *untrusted* inputs into *sensitive* destinations [19]. Generally, a taint analyzer takes four types of inputs (sources, propagators, filters and sinks) to run taint analysis. *sources* are the statements, those generate of untrusted inputs, *propagators* propagate *untrustworthiness* from one variable to another, *filters* purify *untrusted* variables to *trustworthy* ones and *sinks* are the sensitive destinations. Using taint analysis one can ensure the *integrity* property of whether untrusted values can reach and modify trusted placeholders. One may also be interested in the dual property like confidentiality, i.e., whether sensitive values can leak to untrusted sinks. In *static taint analysis*, the taint analysis is performed statically on source code without actually executing it.

In the rest of this section, we discuss how static taint analysis can be used to enforce the prescribed security rules. As *TaintCrypt* is an ongoing effort, instead of extensive evaluation of the prototype, we demonstrate the effectiveness on different coding examples from different vulnerable versions of OpenSSL and Juniper Network.

*Deprecating Vulnerable Functions.* The enforcement of the security rules in Row 1, 5, 6, 10 and 23 of Table 1 demands programmers to avoid/deprecate vulnerable cryptographic functionalities. For these case, one can list vulnerable crypto functions as sources and any other functions those might use the values produces from the listed functions are listed as sinks. If there exists any path between a listed source and its corresponding sink, our *TaintCrypt* engine traces and reports it. In Figure 1, we show how the use of MD5 is detected and warned in OpenSSL[2].

*Ensuring Certain Function Invocations.* To enforce the security rules in Row 2, 9 and 11, we need to ensure the invocation of certain functions. To do that, one can list those functions and use them as filters to trick the taint analysis engine, so that it reports any *dangerous path* from source to sink that

2. This vulnerability existed before commit $f8547f62$

```
951    call_dummy_source_client_random(s->s3->client_random);
       6 ← Expression 's->s3->client_random' gets tainted here →
952    p = s->s3->client_random;
953
954    /*
955     * for DTLS if client_random is initialized, reuse it,
956     * required to use same upon reply to HelloVerify
957     */
958    if (SSL_IS_DTLS(s)) {
       7 ← Taking true branch →
```

(a) Code snippet: source.

```
969
970    if (i && ssl_fill_hello_random(s, 0, p, sizeof(s->s3->client_random))
971        return 0;
1002   *
1003    * For TLS 1.3 we always set the ClientHello version to 1.2 and rely
1004    * supported_versions extension for the real supported versions.
1005    */
1006   if (!WPACKET_put_bytes_u16(pkt, s->client_version)
1007           || !WPACKET_memcpy(pkt, s->s3->client_random, SSL3_RANDOM_SIZ
           11 ← Untrusted data 's->s3->client_random' is passed to this sink!
1008       SSLerr(SSL_F_TLS_CONSTRUCT_CLIENT_HELLO, ERR_R_INTERNAL_ERROR);
1009       return 0;
1010   }
```

(b) Code snippet: filter and sink.

Figure 2: Enforcing secured random number generation in OpenSSL (Row 11). As there exists a path from source to sink that avoids `ssl_fill_hello_random`, our *TaintCrypt* generates a warning.

avoids the filter(s). In Figure 2, we see that the variable `s->s3->client_random`[3] is passed to the sink `WPACKET_memcpy` and `ssl_fill_hello_random` is used as the filter. We see that, there exists at least one path from source to sink that avoids this filter. Since, the existence of such path is unexpected, *TaintCrypt* generates a warning. Close inspection shows that the reported path actually reuses previously generated values. To *whitelist* such special value propagations, one should mark them as filter(s). *Filtering Data From External Sources.* Using data from external sources is unavoidable. However, such data should be filtered/sanitized before use. Hence, to enforce these types security rules (Row 15, 16, 17, 18 and 22), one should list three types of functions: (1) *untrusted* data sources (source); (2) their corresponding sinks (sink); and most importantly, (3) data filters/sanitizers (filter). If there exists any path from source to sink that avoids filter(s) is illegitimate and *TaintCrypt* reports it. In Figure 3, we show that how heartbleed memory disclosure vulnerability can be detected using this methodology, which could be avoided with the incorporation of proper sanitizers.

*Data Leak Detection.* Using taint analysis we can also model the use cases of data leaks such as Row 19 in Table 1. We can list the sensitive data producers as sources and potential mole functions (e.g., function writing data to

3. As the current version of our tool only accepts functions as sources, we use a dummy call on the variable in order to taint it.

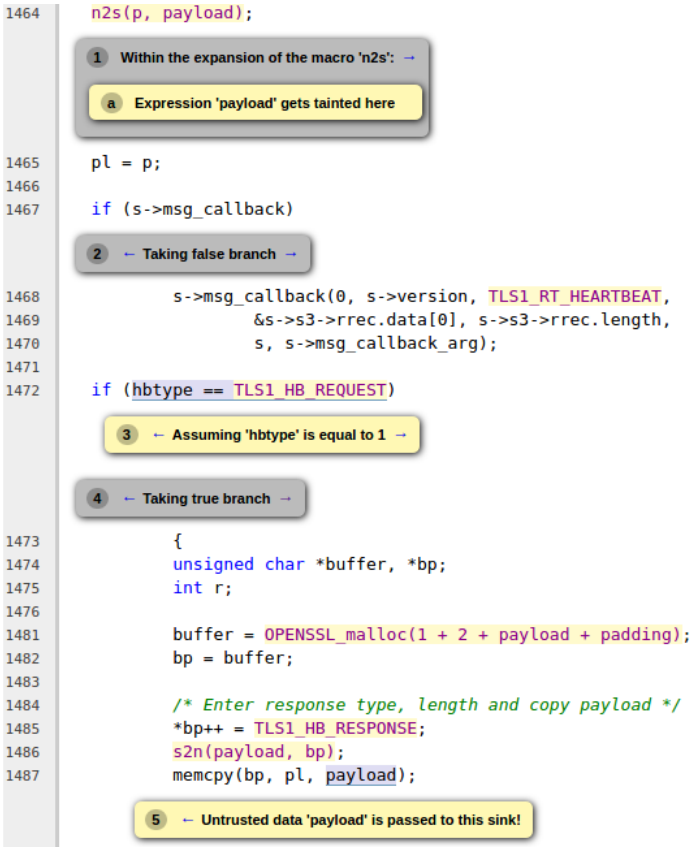
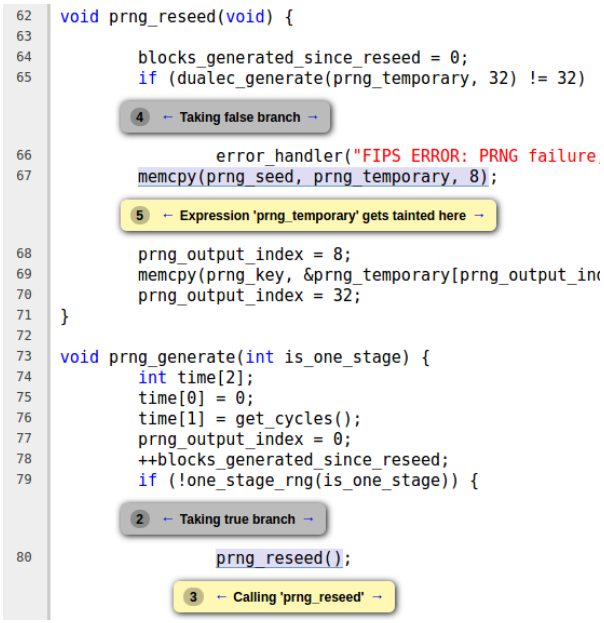
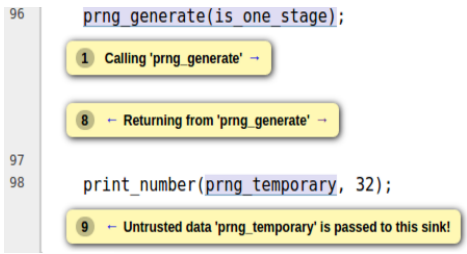



Figure 3: Our *taintCrypt* detects the memory disclosure vulnerability in OpenSSL-1.0.1f (Row 22). Here, the use of `payload` without proper sanitization causes a disclosure of memory of arbitrary size.




(a) Sensitive source.




(b) Untrusted sink.

Figure 4: Our *TaintCrypt* detects and reports the leak of `prng_seed` as the first 8 bytes of `prng_temporary` in Juniper Network (Row 19).

the filesystem or network) as sinks, so that taint analysis engine can detect and report if there exists any path from *sensitive sources* to *untrusted sinks*. In [2], authors showed that ScreenOS of Juniper network was leaking seeds due to programming errors. In Figure 4 we show static taint analysis can be used to prevent such leakage.

*Double Free Detection.* We define the double `free()` incident of Row 12 as follows. If any data is passed through `free()` function, it gets *tainted* and it becomes unsafe to pass into any subsequent invocation of `free()` function. In Figure 5, we show the detection of "double `free()`" in OpenSSL[4].

### 3.3. Limitations

In general, static code analysis offers trade-off among soundness, scalability and false positive rates [53]. Similar to most other static analysis-based approaches, our current prototype has a number of limitations. In our prototype, the symbolic execution based path sensitive analysis takes computationally exponential time [54]. Consequently, the

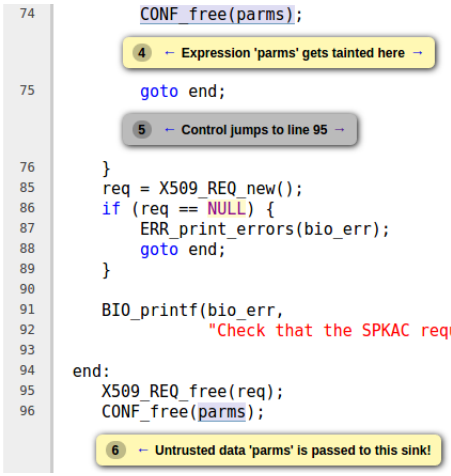



Figure 5: Detection of double free vulnerability in OpenSSL (Row 12). Here, *TaintCrypt* reports the double free incident of `parms` variable.

4. This vulnerability existed till commit $a34ac5b8$

loop unrolling mechanism of the SMT solver used to model symbolic execution is constant bounded. This means soundness of the analysis will cause high computation complexity with high false positive rates. Currently, our taint analysis engine accepts *sources*, *sinks*, *filters* and *propagators* parameters as functions. As a result, in large code bases, the prevalence of constraint-based filters (i.e., the use of *if, else* to screen legitimate inputs from the illegitimate one) might also cause false positives. The soundness of the analysis is also dependent on the coverage of the specified parameters. Lastly, our static analyzer based taint analysis does not work across translational units, currently. As a result, it cannot track taint propagation if the taint sources and taint sinks are located in different translational units.

## 4. Conclusion and Future Work

In this paper, we show that static taint analysis holds the promise to enforce a wide range of security properties in large code bases. We systematically investigate various threat categories on various cryptographic implementations and discuss the capabilities of static taint analysis to enforce various security rules to throttle these threats. By using our prototype implementation, we demonstrate several examples of security enforcement in OpenSSL.

However, our analysis reveals several open problems in this promising line of research. Making *TaintCrypt* work across translational units in a scalable way is still open. In Table 1, we see that taint analysis alone is not sufficient to enforce all the security rules in static program analysis settings. Hence, other mechanisms such as control flow and/or data flow analysis techniques are also needed to be explored to broaden the coverage. Although, memory related errors (e.g., Row 12, 14, 16, 21, 22) are irrelevant for high-level programming languages (*e.g.*, Java, Python, Ruby, JavaScript), still building similar tools for high-level languages will be extremely useful to defend against other misuses of cryptographic libraries or protocols by mass developers. Building language/library specific sensitive *source* and *sink* lists upon which the community can agree upon for consistent evaluation and security guarantees is also interesting to investigate.

## 5. Acknowledgment

We would like to thank Haipeng Cai, Ke Tian, Long Cheng and the LLVM developer community for their support with insightful discussions. We are also thankful to the anonymous reviewers for their helpful comments and suggestions.

## References

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

[2] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the Juniper Dual EC incident. In *ACM CCS 2016*, pages 468–479, 2016.

[3] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: a case study and open problems. In *APSys 2014*, pages 7:1–7:7, 2014.

[4] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *ACM CCS 2016*, pages 1639–1650, 2016.

[5] Sze Yiu Chau, Omar Chowdhury, Md. Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. In *IEEE S&P 2017*, pages 503–520, 2017.

[6] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. HVLearn: Automated Black-Box Analysis of Hostname Verification in SSL/TLS Implementations. In *IEEE S&P 2017*, pages 521–538, 2017.

[7] Andrew Chi, Robert A. Cochran, Marie Nesfield, Michael K. Reiter, and Cynthia 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*, pages 177–195, 2017.

[8] Juraj Somorovsky. Systematic Fuzzing and Testing of TLS Libraries. In *ACM CCS 2016*, pages 1492–1504, 2016.

[9] Joeri de Ruiter and Erik Poll. Protocol State Fuzzing of TLS Implementations. In *USENIX Security 2015*, pages 193–206, 2015.

[10] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering - Design Principles and Practical Applications*. Wiley, 2010.

[11] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: why do Java developers struggle with cryptography apis? In *ICSE 2016*, pages 935–946, 2016.

[12] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *ACM CCS 2013*, pages 73–84, 2013.

[13] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps, 2017.

[14] Karim O. Elish, Xiaokui Shu, Danfeng (Daphne) Yao, Barbara G. Ryder, and Xuxian Jiang. Profiling user-trigger dependence for Android malware detection. *Computers & Security*, 49:255–273, 2015.

[15] Yonghwi Kwon, Brendan Saltaformaggio, I Luk Kim, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. A2c: Self destructing exploit executions via input perturbation. In *NDSS 2017*, 2017.

[16] Kui Xu, Ke Tian, Danfeng Yao, and Barbara G. Ryder. A Sharper Sense of Self: Probabilistic Reasoning of Program Behaviors for Anomaly Detection with Context Sensitivity. In *DSN 2016*, pages 467–478, 2016.

[17] Kui Xu, Danfeng (Daphne) Yao, Barbara G. Ryder, and Ke Tian. Probabilistic Program Modeling for High-Precision Anomaly Classification. In *IEEE CSF 2015*, pages 497–511, 2015.

[18] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In *AsiaCCS 2017*, pages 71–85, 2017.

[19] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS 2007*, 2007.

[20] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective Taint Analysis of Web Applications. In *ACM SIGPLAN PLDI 2009*, pages 87–97, 2009.

[21] Gregory V. Bard. The Vulnerability of SSL to Chosen Plaintext Attack. *IACR Cryptology ePrint Archive*, 2004:111, 2004.

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

[23] Ian Goldberg and David Wagner. Randomness and the Netscape browser. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 21(1):66–71, 1996.

[24] Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko van Someren. Factoring RSA Keys from Certified Smart Cards: Coppersmith in the wild. In *ASIACRYPT 2013*, pages 341–360, 2013.

[25] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX Security 2012*, pages 205–220, 2012.

[26] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual EC in TLS implementations. In *USENIX Security 2014*, pages 319–335, 2014.

[27] David Adrian *et al.* Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *ACM CCS 2015*, pages 5–17, 2015.

[28] Karthikeyan Bhargavan and Gaëtan Leurent. On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN. In *ACM CCS 2016*, pages 456–467, 2016.

[29] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH. In *NDSS 2016*, 2016.

[30] Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In *EUROCRYPT 2002*, pages 534–546, 2002.

[31] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback, 2014.

[32] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *IEEE S&P 2013*, pages 526–540, 2013.

[33] Daniel Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *CRYPTO '98*, pages 1–12, 1998.

[34] Vlastimil Klíma, Ondrej Pokorný, and Tomás 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*, pages 426–440, 2003.

[35] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *CRYPTO 2012*, pages 608–625, 2012.

[36] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's Attack Strikes again: Breaking PKCS#1 v1.5 in XML Encryption. In *ESORICS 2012*, pages 752–769, 2012.

[37] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: breaking TLS Using SSLv2. In *USENIX Security 2016*, pages 689–706, 2016.

[38] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *USENIX Security 2014*, pages 733–748, 2014.

[39] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *ACM CCS 2014*, pages 990–1003, 2014.

[40] David Brumley and Dan Boneh. Remote Timing Attacks Are Practical. In *USENIX Security 2003*, 2003.

[41] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In *ESORICS 2011*, pages 355–371, 2011.

[42] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169, 2002.

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

[44] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time Implementations. In *USENIX Security 2016*, pages 53–70, 2016.

[45] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *IEEE S&P 2015*, pages 535–552, 2015.

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

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

[48] Shuo Chen, Jun Xu, and Emre Can Sezer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security 2005*, 2005.

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

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

[51] 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].

[52] Marcelo Arroyo, Francisco Chiotta, and Francisco 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*, pages 1–12, 2016.

[53] Arnaud Venet and Michael R. Lowry. Static analysis for software assurance: soundness, scalability and adaptiveness. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 393–396, 2010.

[54] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX OSDI 2008*, pages 209–224, 2008.