CipherH: Automated Detection of Ciphertext Side-channel Vulnerabilities in Cryptographic Implementations

Sen Deng
Southern University of Science and Technology

Mengyuan Li
The Ohio State University

Yining Tang
Southern University of Science and Technology

Shuai Wang*
Hong Kong University of Science and Technology

Shoumeng Yan
The Ant Group

Yinqian Zhang*
Southern University of Science and Technology

Abstract

The ciphertext side channel is a new type of side channels that exploits deterministic memory encryption of trusted execution environments (TEE). It enables the adversary with read accesses to the ciphertext of the encrypted memory, either logically or physically, to compromise cryptographic implementations protected by TEEs with high fidelity. Prior studies have concluded that the ciphertext side channel is a severe threat to not only AMD SEV-SNP, where the vulnerability was first discovered, but to all TEEs with deterministic memory encryption.

In this paper, we propose CipherH, a practical framework for automating the analysis of cryptographic software and detecting program points vulnerable to ciphertext side channels. CipherH is designed to perform a practical hybrid analysis in production cryptographic software, with a speedy dynamic taint analysis to track the usage of secrets throughout the entire program and a static symbolic execution procedure on each "tainted" function to reason about ciphertext side-channel vulnerabilities using symbolic constraint. Empirical evaluation has led to the discovery of over 200 vulnerable program points from the state-of-the-art RSA and ECDSA/ECDH implementations from OpenSSL, MbedTLS, and WolfSSL. Representative cases have been reported to and confirmed or patched by the developers.

1 Introduction

The global cloud computing market has experienced exponential growth over the past few years. However, the lack of trust in the cloud operators has always been a major obstacle for many data owners to embrace the cloud technology, especially in finance and health care settings where data security is of concern. To address this trust issue, the concept of confidential computing has been proposed [14], which is a norm of computation that preserves the confidentiality of the computation and data on untrusted computing platforms.

*Corresponding authors. shuaiw@cse.ust.hk, yinqianz@acm.org

The core technique that enables confidential computing is the Trusted Execution Environment (TEE). TEE is a hardware feature available in most modern processors. With the help of a hardware root-of-trust and a fast memory encryption engine, TEE can provide an isolated execution environment for secure data processing, guaranteeing both the confidentiality and integrity of program instances running inside the TEE.

Memory encryption is the primary means to protect memory data against an adversary with either software-level or physical-level access to the memory content. A memory encryption engine is a hardware module sitting in between of the CPU chip and the DRAM module, which automatically encrypt or decrypt the data on the memory bus on-the-fly. The mode of operation used in memory encryption is constrained by two factors: First, to support random memory access in an efficient manner, memory blocks must be encrypted independently; chaining modes (e.g., CBC mode) are not suitable. Second, to support large encrypted memory, encryption modes requiring freshness (e.g., CTR mode) cannot be adopted, as additional space and latency are needed to maintain the counters [33].

As such, all TEEs supporting large encrypted memory, including AMD Secure Encrypted Virtualization (SEV) [30], Intel Software Guard Extensions (SGX) on Ice Lake SP [27,28], Intel Trust Domain Extensions (TDX) [26], and ARM Confidential Compute Architecture (CCA) [4], adopt the AES encryption with *deterministic*, *block-based* modes of operation, such as XOR-Encrypt-XOR (XEX) mode or XEX-based tweaked-codebook mode with ciphertext stealing (XTS) mode. The inclusion of tweak functions—an obscured, one-way function with physical addresses as input—successfully prevents a malicious cloud provider from correlating the memory content at different physical addresses. However, without freshness, the same plaintext block is always encrypted into the same ciphertext block at the same physical address.

Such design choices leads to the notorious ciphertext side channels [33, 34]. It is demonstrated that when the secrets are stored at fixed physical locations (e.g., the VM save area, kernel data structures, user-land stacks, etc.), an adversary having read accesses to the ciphertext (either via software access [34] or via memory bus snooping [32]) may be able to recover some plaintext information of the encrypted memory, leading to complete breach of cryptographic implementations if they exhibit specific patterns of memory updates [33, 34]. We call such a pattern that leads to ciphertext leakage a ciphertext side-channel vulnerability.

While ciphertext side channels were first discovered on AMD SEV, where the hypervisors are granted read accesses to the ciphertext of the encrypted memory, other TEEs are also susceptible to ciphertext side channels, when the adversary is able to perform memory bus snooping. Instead of fixing the vulnerabilities from the hardware, presumably due to the overhead associated, AMD has recommended mitigating ciphertext side channels by fixing ciphertext side-channel vulnerabilities from the software [2].

As the first step of such a software-based solution, in this paper, we propose CIPHERH, a practical framework for automating the analysis of cryptographic software and detecting program points vulnerable to ciphertext side channels. We first formulate ciphertext side channels using symbolic constraints. Our qualitative formulation denotes an adversarial view to check if consecutive secret writes can result in information leakage. Technically, we design CIPHERH to deliver a practical *hybrid analysis* for detecting cipertext side channels in production cryptographic software. CIPHERH launches speedy dynamic taint analysis to track the usage of secrets throughout the entire program. Then, CIPHERH performs static symbolic execution on each "tainted" function (in assembly code), reasoning if secret memory writes induce ciphertext side channels using our formed symbolic constraint. CIPHERH is carefully optimized with respect to both intra-procedural and inter-procedural symbolic analysis, exhibiting greater comprehensiveness and scalability than existing works. We show that existing side channel detectors are not scalable to analyze CIPHERH's test cases.

We have evaluated CIPHERH using real-world cryptographic libraries and achieved encouraging results. Within 28 CPU hours, CIPHERH is able to complete the analysis of RSA and ECDSA/ECDH implementations from three widely-used cryptographic libraries, OpenSSL (ver. 3.0.2), MbedTLS (ver. 3.1.0), and WolfSSL (ver. 5.3.0). CIPHERH successfully detects 236 vulnerable program points, among which only nine are false positives according to our manual confirmation. Representative findings of CIPHERH have been responsively confirmed by OpenSSL and WolfSSL developers, and some patches have been issued. Particularly, we show surprising findings that CIPHERH detects new vulnerabilities in WolfSSL that can exploit newly patched code. WolfSSL developers confirmed our findings and prepared another patch. In sum, our contributions are as follows:

- We formulate for the first time ciphertext side channels, an emerging threat on mainstream TEEs, using symbolic constraints. This enables automated analysis to determine if consecutive secret writes lead to ciphertext leakage.
- We design CIPHERH, a framework identifying ciphertext side channels in production cryptographic software. CIPHERH explores a synergistic effect by combining scalable whole-program dynamic taint analysis and precise function-level static symbolic execution. It outperforms existing side channel detectors with higher scalability and comprehensiveness.
- We have applied CIPHERH to analyze popular cryptographic software and discovered a large number of vulnerabilities. Representative cases have been confirmed by the developers, and CIPHERH even finds new vulnerabilities from recently patched code in WolfSSL.

2 Background of Ciphertext Side Channels

Li *et al.* [34] first discovered ciphertext side channels in AMD SEV-SNP. While other attacks against AMD's TEE [35–37, 40, 55] were already fixed in the latest variants of SEV (SEV-SNP), the ciphertext side channel is the only one that exploits a design flaw in SEV-SNP. Due to the use of deterministic, block-based AES encryption, ciphertext changes of heaps, stacks, or kernel memory in the SEV guest VMs could leak information about execution states and even secret keys [33].

Deterministic Encryption. Deterministic encryption are widely used in TEEs with large encrypted memory, such as AMD SEV [30], Intel TDX [27], Intel SGX on Ice Lake SP [27, 28], and ARM CCA [4]. Most TEEs may use AES-XEX and AES-XTS modes of operation. For instance, in XEX, for encrypting a 128-bit memory block in physical address P_m, MEE first uses P_m to calculate a tweak value $T(P_m)$. The plaintext m in this memory block is then XOR-ed with $T(P_m)$ before encryption. The output of the encryption is later XOR-ed with $T(P_m)$ again to generate the final ciphertext c. Therefore, the ciphertext is $c = ENC(m \oplus T(P_m)) \oplus T(P_m)$. Deterministic encryption with either XEX or XTS mode of operation leads to ciphertext side channels, because the same plaintext at the same physical address is always encrypted into identical ciphertext.

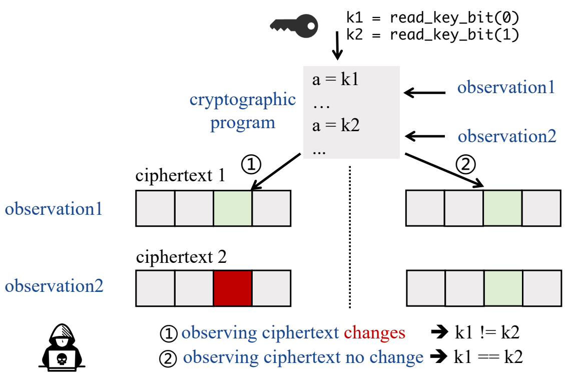

Figure 1: Inferring relations of key bits.

Dictionary and Collision Attacks. Ciphertext side-channel attacks are classified into *Dictionary Attacks* and *Collision Attacks* [33]. Dictionary attacks are attacks where the adversary collects sufficient ciphertext-plaintext pairs at fixed physical address and builds dictionary, which are later used to infer the plaintext when the corresponding ciphertext is observed. Collision Attacks are attacks where memory write behaviors directly leak secret-related information. Figure 1 presents a schematic view of launching collision attack, where `k1` and `k2`, denoting the first and the second bits of the private key, are consecutively written to variable `a`. By observing if the ciphertext is changed or not after the second memory write operation, the attacker can easily infer the relations (equality/inequality) of key bits.

Countermeasures. Hardware-based countermeasures, such as changing the encryption mode or preventing read accesses to encrypted memory, may be implemented but with higher performance cost. In contrast, software-based countermeasures that alter the operating systems and cryptographic libraries are recommended by AMD [2]. Li *et al.* [33] proposes several mitigation strategies and some have been incorporated into WolfSSL. However, as described in §7.2, even patched code may to contain subtle leaks. CIPHERH is the first tool that offers a systematic and automated analysis of cryptographic software under the cipertext side-channel threats. Moreover, CIPHERH can assist the evaluation of the efficacy of countermeasures in production software.

3 Threat Model and Application Scope

Threat Model. In this paper, we follow the established threat model of TEE: the adversary is assumed to have full system privilege on the machine and is also capable of performing physical attacks, including inferring address and content of every memory read via memory bus snooping [32], reading remnant data from the DRAM via cold boot attack [24], and accessing memory directly via DMA devices [47].

We assume the targets of attacks are the secrets in the cryptographic software that runs inside a VM TEE, such as SEV-SNP. The software stack inside the VM, including the OS and application, is secure, such that the adversary cannot alter its control flow or force it to leak secrets voluntarily. We assume the hardware and microcode of the processor is up-to-date: known attacks against SEV, SEV-ES, and SEV-SNP [29] have all been fixed, leaving only generalized ciphertext side channel leakage discovered in Li *et al.* [33].

Application Scope. Recent works have illustrated the feasibility of launching ciphertext side-channel attacks against cryptographic libraries using the aforementioned threat model [33]. However, they primarily target manually identified program vulnerabilities. This work presents CIPHERH, a thorough and fully automated framework for identifying such vulnerabilities in production cryptographic libraries. As reported in §7, CIPHERH successfully detects a large collection of program points vulnerable to ciphertext side channels in prominent cryptographic libraries, including program points recently patched against ciphertext side channels.

We clarify that the main audiences of CIPHERH are developers who want to deploy their cryptographic software on modern TEEs. Before release, CIPHERH serves as a "vulnerability debugger" to assist developers in assessing potential attack vectors of their software under ciphertext side channels. CIPHERH provides fully automated and systematic analysis to flag program points that leak secrets via ciphertext side channels. Developers can accordingly patch CIPHERH's findings to mitigate leakage. This design decision is consistent with the majority of side-channel detectors in this field (though they target cache side channels rather than ciphertext side channels) [6, 10, 13, 16, 19, 48, 51, 52, 54, 56]. Nevertheless, we clarify that CIPHERH is *not* an attack tool; the exploitability of its findings (e.g., whether RSA private keys can be reconstructed via CIPHERH's findings) is beyond the scope.

Also, as introduced in §2, ciphertext side channels are currently only studied and performed on AMD SEV because other TEEs with large memory (*e.g.*, ARM CCA and Intel TDX) are not commercial available yet. We note, however, that CIPHERH is *not* exclusive to AMD. The analysis of CIPHERH is orthogonal to the target platform and can be applied to examine leakage from cryptographic software on any other deterministic encryption-based TEE architectures. The proposed steps in CIPHERH can also serve as a reference approach to help check leakage in applications other than cryptographic libraries.

4 Modeling Ciphertext Side Channels

This section formulates ciphertext side channels using symbolic constraints. Then, with the help of constraint solvers, we are able to rigorously detect if secret data written to memory can be leaked via ciphertext side channels. Our modeling in this section primarily considers *collision attack*, as a more general form, which can naturally subsume dictionary attacks.

Notation. We use $W(k)$ to represent the memory written value, where $W(k)$ is a symbolic expression (containing k) that denotes the written value and k is a free variable representing secrets. Such function-like notations are also used in the literature [6, 51, 52], although in this paper $W(k)$ is the memory written value (depending on secret k), whereas in previous cache side channel papers the formula (typically written as $F(k)$) denotes a memory address based on the secret.

In collision attacks, we consider two sequential write operations toward the *same memory location*, whose written values are $W_1(k_1)$ and $W_2(k_2)$ ($W_2(k_2)$ occurs after $W_1(k_1)$). Note that k_1, k_2 may denote the same or different sub-keys. For instance, considering our illustrative example in Figure 1, we use $W_1(k_1) = k_1$ and $W_2(k_2) = k_2$ to encode written values of two consecutive memory writes toward variable `a`.

Two Safe Cases. Before presenting the symbolic constraint that encodes ciphertext side channels, we first introduce two *safe* cases that are free from ciphertext side channels.

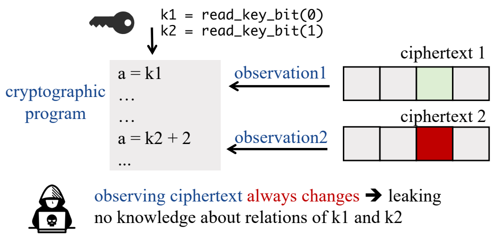

Figure 2: No leakage as the ciphertext *always* changes.

Safe Case 1: Following Figure 1, Figure 2 illustrates our first safe case. Note that `k1` and `k2`, each of which denotes one key bit, could be 0 or 1. Therefore, $\texttt{k2}+2 \geq 2 \neq \texttt{k1}$ shall always hold, meaning that from the attacker's perspective, the second memory write operation *always* leads to ciphertext change, and therefore, she can learn nothing about the relations of `k1,k2`. In sum, we deem this as a safe case, which can be expressed using the following symbolic constraint:

$$\forall k_1, k_2 \in K, W_1(k_1) \neq W_2(k_2) \qquad (1)$$

such that despite the value of k_1, k_2 in K where K denotes all possible values of secrets, the memory write contents $W_1(k_1)$ and $W_2(k_2)$ should always be distinct.

Safe Scenario 2: Similarly, considering the following case,

$$\forall k_1, k_2 \in K, W_1(k_1) = W_2(k_2) \qquad (2)$$

This constraint implies that despite the value of k_1, k_2, the memory write contents $W_1(k_1)$ and $W_2(k_2)$ are *always the same*. This is another safe case, as from the attacker's perspective, the observed ciphertext stays constant, leaking no information about the relations of k_1, k_2.

Information Leakage Case. We underline that information leakage occurs when neither of the two safe cases are satisfied. In other words, the "unsafe" case needs to model *two different executions* following the same path (we discuss path constraint later in this section), such that during one execution, the second memory write operation changes ciphertext, whereas during the other execution, the second memory write operation retains the ciphertext. To model two executions in a single constraint, we take a common tactic, self-composition [16,52], such that we self-compose a formula with a renamed version of itself. In particular, Considering the following constraint,

$$\exists k_1, k_2, k'_1, k'_2 \in K, W_1(k_1) = W_2(k_2) \wedge W_1(k'_1) \neq W_2(k'_2) \quad (3)$$

where we apply self-composition to rename all occurrences of k_1 in W_1 with free new symbol k'_1 to obtain $W_1(k'_1)$, and similarly for k_2 in W_2. This constraint is sent to a constraint solver. If the solver returns "SAT", it means that the observed ciphertext *depends* on the values of secret k_1, k_2. Particularly, when the observed ciphertext stays the same among two memory writes, the attacker can establish relations $(W_1(k_1) = W_2(k_2))$ between k_1 and k_2. When observing the ciphertext changes, she can exclude relations $(W_1(k_1) = W_2(k_2))$ between k_1 and k_2. These conclude the leakage of program secrets via ciphertext side channels.

Path Constraint and Public Data. The solution of constraint presented in Eq. 3 may not be feasible in practice, because program execution may not cover these two memory accesses. As a common strategy, we extend Eq. 3 with path constraint C. C is a logic formula, denoting the conjunction of all branch conditions from the program entry point to the second program memory write (which also covers the first memory write instruction alone the execution). Also, when performing symbolic execution, public input and public local data are also represented using public symbols (omitted in Eq. 3). When performing self-composition to generate Eq. 3, we follow the standard practice to constraint all public symbols to be equal in both executions [52]. Thus, if C contains only public symbols, two executions modeled in Eq. 3 are guaranteed to follow the same path. And if C contains secret symbols, each secret symbol k is renamed during self-composition, and the constraint solver will search for secrets used by two executions that satisfy both the path constraint and Eq. 3. In short, we check the following augmented constraint:

$$\exists k_1, k_2, k'_1, k'_2 \in K, W_1(k_1) = W_2(k_2) \wedge W_1(k'_1) \neq W_2(k'_2) \wedge C \qquad (4)$$

```
1. for (i = cardinality_bits - 1; i >= 0; i--) {
2.   kbit = BN_is_bit_set(k, i) ^ pbit; //read one bit of k
3.      EC_POINT_CSWAP(kbit, r, s, group_top, Z_is_one);
4.      ...
5.   pbit ^= kbit; // vulnerable memory write using kbit
```

Figure 3: CIPHERH running example.

Working Example. Consider Figure 3, which illustrates the usage of the formed constraint in Eq. 4. This code snippet is from the ECDSA Montgomery ladder algorithm in OpenSSL. In each loop iteration, one bit of the secret k is fetched via `BN_is_bit_set`, determining a conditional swap at line 3. The vulnerability is at line 5, where `kbit` is first calculated with the value in `pbit` using exclusive or (xor), and then written to `pbit`, a variable on the stack.

Let symbol k_i (secret symbol) denote the return value of `BN_is_bit_set()` in the ith loop iteration and symbol a (public symbol) denote the variable `cardinality_bit`. Given that lines 2 and 5 perform xor operations with `pbit` twice, its effect on the memory write at line 5 is canceled out. Also, the function call at line 2 does not modify `kbit`. Thus, when performing symbolic execution, we derive two formulas, $W_1(k_1) \equiv k_1$, and $W_2(k_2) \equiv k_2$, for two consecutive memory writes (occuring during consecutive loop iterations) at line 5. To check if ciphertext side channels exist, we perform self-composition to rename k_1, k_2 in W_1, W_2 with k'_1, k'_2, and get $W_1(k'_1) \equiv k'_1$, and $W_2(k'_2) \equiv k'_2$. According to the loop condition at line 1, the path constraint C for the second memory write is "$a-1 \geq 0 \wedge a-2 \geq 0$". Since there are no secret symbols in C, the patch constraint is retained. Hence, we check the satisfiability of the following constraint according to Eq. 4:

$$k_1 = k_2 \wedge k'_1 \neq k'_2 \wedge a - 1 \geq 0 \wedge a - 2 \geq 0$$

The constraint solver yields SAT (meaning the constraint is solvable), and provides the following solution such as:

$$[k_1 = 0, k_2 = 0, k'_1 = 0, k'_2 = 1, a = 2]$$

To interpret the results, we show that two consecutive memory writes at line 5 will result in different ciphertext changes, depending on the value of k_1, k_2. When attackers observe the ciphertext does not change in the first two loop iterations, she infers that $k_1 = k_2$, and verse versa. This program point leaks substantial sensitive information about the (in)equality of every two consecutive secret bits, and CIPHERH can automatically find it when analyzing the codebase of OpenSSL.

Leakage Encoding Soundness/Completeness. §2 introduces real-world collision attacks [33] that exploit consecutive memory writes to the same location. Accordingly, our threat model (§3) and leak encoding (Eq. 4) assume that successive memory writes are toward the same memory address *addr*. To ease presentation, we assume that *addr* only depends on public information when introducing the leakage encoding in this section. Nevertheless, when *addr* is secret-dependent, satisfiable solutions of Eq. 4 may refer to different store operations. This illustrates the *incompleteness* (false positives) of Eq. 4. However, we do *not* observe such subtle false positives when analyzing real-world cryptographic software, which strives to be constant-time.

Furthermore, our leakage encoding is *unsound*. Note that in addition to collision attacks launched to exploit ciphertext side channels [33], we envision that secret-dependent control branches/memory addresses may also be exploited via ciphertext side channels. Different secret values, for instance, may result in executing distinct secret-dependent branches and updating different ciphertext blocks. Our leakage encoding does not consider such cases due to two reasons: 1) detecting secret-dependent control branches/memory addresses is technically feasible by extending de facto cache side channel detectors, and 2) we are not aware of any real-world ciphertext attacks that rely on such dependencies. Leveraging such dependencies to exploit and mitigate ciphertext side channels is left for future research.

Design Consideration: Information Flow Tracking. When forming the constraints in this section, we assume that k_1, k_2 represent program secrets. Nevertheless, in addition to the ciphertext side channels directly over the secrets, it is crucial to treat data derived from the secrets as "sensitive" and tracking information propagation [45]. As a common practice in this line of research, we design CIPHERH to track explicit and implicit information flow propagated from the secret. This shall comprehensively model the attack surface of cryptographic software.

Design Consideration: Verification vs. Detection. Formulations in this section reveal the feasibility of rigorously verifying the absence of cipertext side channels: we can leverage static analysis techniques like abstract interpretation [15] to obtain a sound approximation (the "upper-bound") of program semantics at each memory access point, and check the satisfiability of Eq. 2 and Eq. 1 to prove the safety of the program. Nevertheless, performing whole-program abstract interpretation is inherently not scalable. For instance, relevant works performing abstract interpretation on binary code [19, 20, 51] either analyze toy programs [19] or manually-scoped sensitive code fragments [20, 51].

Hence, we instead aim to design a scalable bug detector for cryptographic software. As will be introduced in §5, CIPHERH explores a *hybrid* approach by using dynamic taint analysis (which is scalable and rapid) to collect functions tainted on an execution trace. Then, it performs static symbolic execution toward each tainted function, covering paths that are not on the dynamic trace. CIPHERH is more comprehensive and accurate than dynamic taint analysis, as it analyzes different paths in a "tainted" function, and uses constraint solving to detect ciphertext side channels. CIPHERH, in addition, is much more scalable than whole-program static analysis, as it mostly perform intra-procedural static analysis (see §5.2.1) toward tainted functions, whereas conventional static methods perform inter-procedural analysis toward all functions. CIPHERH can also perform lightweight inter-procedural analysis to harvest more vulnerabilities; see our optimized strategy in §5.2.2.

Design Consideration: Constraint Solving. It is worth noting that the constraint solver can provide a pair of counterexamples k_1, k_2 that satisfy Eq. 4 (meaning we find a ciphertext side channel). This counterexample can be used to debug the cryptographic software, making it easier to expose the vulnerability. In fact, we manually confirm some CIPHERH's findings by debugging the cryptographic software with counterexamples in §7.1.

Design Consideration: Standard vs. Relational Symbolic Execution. CIPHERH performs standard symbolic execution and self-composition to generate Eq. 4. As noted in recent papers [16], this standard approach is less scalable for two reasons. ① Self-composition creates numerous (unnecessary) queries independent of secrets. ② Self-composition duplicates the entire formula (as in Eq. 4), doubling the size of the self-composed formula and imposing a heavy burden on constraint solving. CIPHERH alleviates ① by checking if secret symbols (k_i) exist in the formula before self-composition. Also, CIPHERH launches only function-level symbolic execution, with "empty" symbolic states at each function entry point. This design reduces constraint solving cost, but may introduce false positive and false negative findings (see §9). Relational symbolic execution (RelSE) [21, 43] can principally eliminate ① and ②, and its recent extensions [16, 17] are carefully optimized to apply RelSE toward binary code. We leave adopting RelSE as a future work to improve scalability.

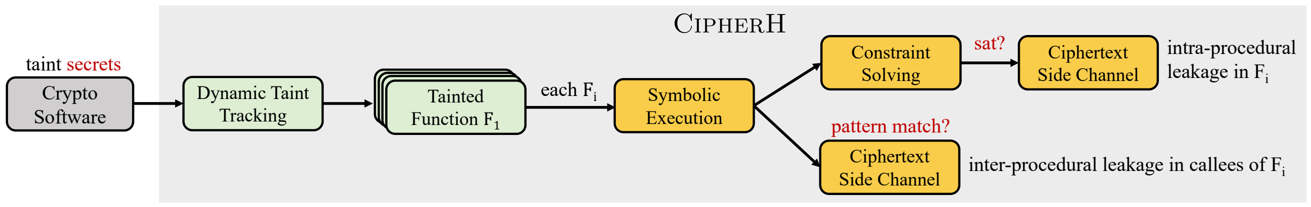

Figure 4: The analysis workflow of CipherH.

5 Design

As noted in §4, when analyzing modern cryptographic software, performing whole-program static symbolic reasoning is prohibitively expensive, if not impossible. Therefore, the technical pipeline of CipherH explores a *synergistic effect*, by combining whole-program dynamic taint analysis and per-function static symbolic reasoning.

Dynamic taint analysis enables coarse-grained and efficient tracking of how secrets are used in cryptographic software. In contrast, static symbolic execution, a more costly technique, is conducted toward each path of a "tainted" function. It delivers a finer-grained, constraint solving-based detection of ciphertext side-channel vulnerabilities.

Figure 4 depicts the overall workflow of CipherH, which consists of a dynamic analysis phase and a static analysis phase. Given a target cryptographic software, CipherH first performs a dynamic taint analysis to analyze how program secrets ("tainted" in this phase) are used by functions in the software (see §5.1). When executing real-world cryptographic software, thousands of functions are frequently invoked in the call chain. This dynamic analysis phase delivers a rapid and scalable analysis to track the potential usage of secrets. Its output is usually a small collection of "tainted functions."

The static analysis phase of CipherH is performed on the executable E compiled from the cryptographic software. This phase has two stages (§5.2): intra-procedural symbolic reasoning and inter-procedural analysis. The intra-procedural stage analyzes each tainted function $F_i \in E$: we launch symbolic execution to analyze every path in F_i, and for a memory write of secrets, we form a constraint according to §4 to check for ciphertext side channels. The inter-procedural stage performs scalable albeit coarse-grained symbolic reasoning over F_i and its callee functions to check if a callee function of F_i satisfies our cipertext leakage patterns (see §5.2.2). If so, CipherH reports F_i and the flagged callee function.

Compiler-IR Taint Tracking. CipherH first performs taint tracking about the usage of program secrets. Taint analysis has been studied widely in the literature. In our research, we also explored the direction of binary code-level taint analysis. Nevertheless, our preliminary study shows that speed is a primary limit for binary-level solutions. Moreover, we find that the information flow policies in binary-level tainting tools, particularly for implicit information flow, appear to be less comprehensive. Some works use trace-based approach, such that an execution trace will be first logged, and taint analysis is performed on the trace. However, those trace-based taint analysis is inherently confined to analyzing defects on a single trace, which likely cause false negatives (see cases in §7.4).

Since CipherH is primarily meant for cryptographic software developers, we explore compiler-level solutions. Recent relevant research [9] has demonstrated the applicability of DataFlowSanitizer (DFSan) [38], a dynamic taint tracking solution provided by the LLVM ecosystem, for taint tracking in real-world cryptographic software. Consequently, we also use DFSan for dynamic taint analysis; see details in §5.1.

Binary Code Symbolic Execution. Aligned with most existing works in this field [6, 16, 19, 51, 52, 56], CipherH detects ciphertext side-channel vulnerabilities based on binary instructions (after obtaining the taint analysis results). This design decision can better take into account low-level details like register allocation and memory accesses. In §7.6, we show that compiler optimizations can affect the presence of ciphertext side-channel vulnerabilities, an observation also noted in [46] (though they focus on cache side channels). CipherH currently uses a well-developed binary analysis framework, angr [5], for symbolic execution. In the static analysis phase, angr needs to disassemble the executable of cryptographic software. Our observation shows that angr manifests a high engineering quality for these reverse engineering tasks. We thus assume that the disassembling and analysis results of angr are *reliable*. Complex executables such as obfuscated code are beyond the scope of our research, as CipherH is mainly designed for developers to test their own software.

5.1 Dynamic Taint Analysis

We now elaborate on the dynamic analysis phase of CipherH. The input of CipherH is cryptographic software S, a set of secrets labelled by the user, and a set of well-formed inputs to S. As a common assumption shared by most existing trace-based side channel detectors [6, 52], well-formed inputs shall smoothly cover the primary key usage procedure (e.g., decryption) in cryptographic software. More importantly, CipherH compensates for uncovered paths via static analysis when these uncovered paths can be reached starting from a function covered in the dynamic phase (see §5.2).

DFSan implements a comprehensive set of information flow rules to track information propagation across variables and memory regions. We confirm that DFSan faithfully tracks program explicit information flow across arithmetic, logic, and memory operations. Moreover, it tracks implicit informa-

```
uint32 x, y, key[3]; // key is secret
uint32 buf[12];
if (key[0] == 12) // secret-dependent control flow
  x = 1;          // implicit flow taint x

y = buf[key[1]];  // secret-dependent mem access
                  // implicit flow taint y
```

Figure 5: Two implicit information flows captured by DFSan.

tion flow occurred at secret-dependent control-flow branches and memory accesses. Figure 5 illustrates these two implicit information flows. In particular, `x` is tainted, given that the first element in the secret `key` is used in the if condition. Furthermore, `y` is tainted, given that the index of `buf` is derived from the second element in `key`. In addition, DFSan correctly tracks inter-procedural taint propagation, in the sense that tainted inputs in a function callsite will be propagated to the callee function parameters, and the tainted return value will be propagated back to the caller function.

Taint Propagation and Tainted Function Collection. To use DFSan, we compile the cryptographic software S into LLVM intermediate representation (IR), which inserts necessary routines alongside each IR statement to track the propagation of taint labels. We use the DFSan APIs to initialize the taint labels at user-marked program secrets. These program secrets are the taint source points. Accordingly, we use DFSan APIs to hook each function entry point, memory load, and function callsite as a taint sink point. Then, we compile and execute the instrumented IR code with user-provided test input and record which taint sink point is tainted during execution. This phase will produce a collection of "tainted" functions, with each function satisfying one of the following conditions: 1) any of its input is tainted, 2) the return value of its callee function is tainted, and 3) it loads tainted data from memory. These tainted functions are candidates of the static analysis phase, as explained below.

Input Requirement. Typically in cryptographic libraries, legitimate inputs can cover primary computation procedures (e.g., decryption). We find that in comparison with general-purpose software, coverage is less concerned when performing dynamic analysis toward cryptographic libraries. Changing different ciphertext (or private keys) does not notably change the execution flow; some loop statements may be performed for more (or less) iterations in accordance with different ciphertext. Overall, the taint analysis phase requires "legitimate inputs", meaning reasonable, common test inputs. When performing taint analysis, we only log one execution trace (with one input) for each case (e.g., the RSA implementation in WolfSSL). CIPHERH does not need many inputs to reach functions that process secrets.

5.2 Static Symbolic Reasoning

Our symbolic execution targets tainted functions collected during the dynamic taint analysis phase. We perform both intra- and inter-procedural symbolic analysis to identify program points vulnerable to ciphertext side channels. We now discuss the intra-procedural and inter-procedural analyses.

5.2.1 Intra-Procedural Symbolic Reasoning

Given the executable file e compiled from the cryptographic software, we use angr to disassemble e and analyze the assembly functions. Note that for each tainted function f_s, the taint analysis phase has flagged certain tainted data in f_s, which may be function parameters, data loaded from memory, or the return value of a function callsite. The intra-procedural symbolic execution is mostly standard: we follow the conventional setup to create fresh symbols s_i to represent the unknown register or memory cell values. Moreover, we create new "key symbols" (e.g., k_i, as used in our constraints in §4) to represent program variables or memory cells that are tainted, i.e., they are part of the program secrets or are derived from them via information flow propagation.

While tainted function parameters and return values can be easily marked in assembly code, it is more obscure to mark tainted data loaded from memory, as our taint analysis occurs in LLVM IR rather than assembly code. We thus take a conservative design, and create new key symbols for any data loaded from heap, globals, or caller functions' stack frames, as long as one tainted data is loaded from memory. This is a conservative design decision, as we "over-taint" heap/globals/previous stack frameworks without performing point-to analysis on assembly code (which is hardly possible) to decide precise usage of secrets. Moreover, given our symbolic execution covers many paths not covered by the taint analysis phase, this "over-tainting" design helps to better estimate the usage of secrets within different paths and detect ciphertext side-channel vulnerabilities.

After initialization, we then launch static symbolic execution from the entry point of f_s to traverse its control flow graph in a path-by-path manner. At each *if* condition, angr constructs symbolic constraints of the encountered branch condition and finds solutions that satisfy either the true or false branch. This way, we achieve much better intra-procedural code coverage than trace-based analysis. See how static symbolic execution enables uncovering vulnerabilities that are not covered by taint analysis in §7.4. When encountering a callsite, we refrain from analyzing the callee function, but create a new symbol denoting the unknown value of the function call returns. Note that in case this return value was tainted during our taint analysis, we create a new key symbol k_i, whereas we create a non-secret symbol s_i for others. As an essential component of symbolic execution, CIPHERH maintains a lookup table $\mathcal{M}$ during the intra-procedural traversal. Each item $(a, v) \in \mathcal{M}$ denotes a value that has been written to memory, where a is the symbolic memory address and v is the symbolic value. For each memory access via address a', CIPHERH searches for the existence of a' in $\mathcal{M}$ (this involves "pointer alias" analysis; details will be discussed below). Depending on whether the memory access is read or write, CIPHERH fetches or modifies the entry's content if the search returns a match. If not, CIPHERH inserts a new entry for a' in $\mathcal{M}$ containing a fresh symbol v' denoting unknown value read from memory, or the value to be written for memory write.

Deciding Pointer Alias. When performing a memory access toward the memory lookup table $\mathcal{M}$ using a symbolic pointer a', we need to decide if the address a' has been previously recorded in $\mathcal{M}$. This step involves a comparison between a' and existing addresses $(a, \_) \in \mathcal{M}$.* This task is particularly difficult for symbolic execution, given that many pointers are represented in the symbolic forms, e.g., we need to decide if a pointer $4a+8$ equals to another pointer $8a+4$.

Determining if two symbolic pointers are equivalent ("alias") is challenging. Angr compares two symbolic pointers based on their syntactical equivalence, which is also adopted in relevant static analysis of side channels [51]. In the current implementation of CIPHERH, we adopt angr's solution to compare the syntactical equivalence of two pointers. This strategy is accurate, because if two pointers are syntactically equivalent, then they must point to the same memory location. Nevertheless, it may potentially overlook some pointers that are syntactically distinct but semantically equivalent. In fact, our tentative study explored to extend angr and to decide the semantic equivalence of two symbolic pointers using constraint solving techniques. For instance, constraint solvers can rigorously prove that two pointers $a+4$ and $a+8$ must not be equivalent (a is a base address of an array), whereas pointers $4a+8$ *may be* aliased to $8a+4$. This approach, however, is very slow in practice, because a large volume of constraint checking is involved when analyzing complex software like production cryptographic libraries. Overall, we deem the current solution of deciding pointer alias as a practical tradeoff which offers a high level of accuracy and scalability.

Checking Ciphertext Side Channels. To ease forensics, CIPHERH maintains a separate memory write table $\mathcal{W}$ similar to $\mathcal{M}$ explained before. Each item $(a, v, i) \in \mathcal{W}$ denotes a symbolic value v that has been written to memory address a by assembly instruction i. When a memory write via address address a' and content v' occurs at instruction i', we check whether v', as a symbolic formula, contains key symbols. If so, we further compare a' with each $(a_i, v_i, i_i) \in \mathcal{W}$ to decide if there exists a_i satisfying $a_i = a'$. We also check if v_i, denoting the most recently-written data to a', contains key symbols. Note that here we check the syntactical equivalence to decide if a_i and a' are alias. If so, we feed v_i and v' to form the cipertext side channel constraint (Eq. 4) and check the satisfiability. This way, we can decide if ciphertext side channels exist at i'. When Eq. 4 is satisfiable, CIPHERH returns the vulnerable program points i', i_i and a pair of counterexamples to users for debugging. CIPHERH also updates the value stored in a' with v' and resume symbolic traversal.

*The strategy of updating the lookup table $\mathcal{M}$, representing memory, is often referred to as a "memory model" [11, 12, 23].

5.2.2 Inter-Procedural Symbolic Reasoning

Motivation. The above intra-procedural analysis enables the discovery of a substantial number of ciphertext side channels, as reported in §7. Nevertheless, we clarify that a memory address a may be repeatedly written using secrets across multiple function calls. In fact, §7.2 will report a surprising finding such that "inter-procedural" ciphertext leakage enables exploiting a recent patch of ciphertext side channels in WolfSSL. Therefore, we envision it as demanding to perform inter-procedural analysis and better reveal the full attack interface of cryptographic libraries under ciphertext attacks.

Challenge. Having that stated, our preliminary study illustrates that standard inter-procedural analysis of modern cryptographic systems is hardly possible owing to a lack of scalability. As discussed in §4, though recent works launching static analysis toward cryptographic libraries [19, 20, 51], they only handle very small programs or crucial code fragments (several caller/callee functions) in cryptographic libraries [20, 51]. As will be shown in §7, both CacheS [51] and CacheAudit [19, 20] fail to analyze full call graphs of our evaluated cryptographic libraries.

Pattern-Based Search. This work takes a reasonable tradeoff for inter-procedural analysis. In particular, we search for function F that is suspicious for ciphertext side channels, if F meets both pattern ① and ② described below. ① F is repeatedly called by its caller function F_c at the same callsite. This means that F is called in a loop of F_c. ② At least one of f's input parameters is tainted. Both ① and ② are are owning to observations over real-world cryptographic software, in which function F_c calls its callee function F, often denoting a small routine function, and passes secrets to F via parameters. By repetitively calling F, F_c processes consecutive fragments of a secret key (or encrypted data). This assumption helps to eliminate many false positives, as shown in §7.5.

In all, during our intra-procedural analysis, we use symbolic execution to traverse each path of a tainted function F_c. During the traversal, we search for every encountered callee function and decide if any callee function F meets the aforementioned patterns. If so, we will proceed further to launch intra-procedural symbolic execution toward F, and decide if F has memory write instructions whose content is derived from its tainted parameters. If so, we would conclude that F enables ciphertext side channels, assuming multiple runs of F from F_c write secrets into the same memory address.

Clarification. To clarify, this inter-procedural analysis may introduce *false positives*, as consecutive runs of F in a loop may have different non-secret inputs or load different contents from memory, and so may not follow the same path and memory access footprints. Besides, even with presence of secret memory writes, the ciphertext side channel constraints may be indeed unsatisfied (as we do not perform constraint solving), leading to false positives. It may also introduce false negatives, as obviously cross-function ciphertext side channels may occur without necessarily following our defined patterns. It is possible that secrets are passed via memory loads or return values of F's callees, instead of F's parameters. Overall, our inter-procedural analysis is primarily designed with *scalability* and observations over real-world cryptographic software's implementation into mind. Though inaccurate, we show that, in practice, the proposed patterns achieve high scalability and reasonable accuracy, detecting a substantial number of true positive findings (after manual confirmation) with negligible false positives (9 out of 153); see details in §7.

6 Implementation

CIPHERH contains about 2.3k lines of code, including a taint analysis module written in C++ by extending LLVM DFSan, a symbolic execution module implemented over angr in Python, and some scripts.

Taint Analysis. We compile each cryptographic software using gllvm [22], which can generate whole-program LLVM bitcode files from unmodified C/C++ source code. Dynamic taint analysis is performed through DFSan. We write taint source configurations to taint private keys in the evaluated cryptographic software. We also define taint sink points as a function's parameters, return values of its callsites, and memory load outputs. DFSan requires compiler instrumentation for all code, except for functions listed in the ABI list. After instrumenting LLVM IR with DFSan utilities, we compile the bitcode file into an executable. To execute each cryptographic software, we use their shipped test programs or write simple test programs. These test programs contain reasonable inputs to invoke RSA encrypt/decrypt, ECDSA sign/verify, and ECDH key exchanges procedures in the evaluated software.

Symbolic Execution. When performing symbolic execution, we compile the cryptographic software into a 64-bit ELF binary executable. The call graph is first constructed in angr to get information about caller/callee relations. Such information is used in the inter-procedural taint analysis phase (§5.2.2). When analyzing each tainted function F_i, we first note all tainted variables in F_i identified during the taint analysis phase, which can be function parameters, return value of its callee, or memory load operations. We create a fresh key symbol k_i to represent the value of each tainted variable. Angr uses a popular constraint solver, z3 [18], to check each formed constraint during the intra-procedural analysis phase. As reflected in §5, CIPHERH improves angr by disabling its default cross-function analysis, and implement our own inter-procedural analysis procedure (§5.2.2). CIPHERH also enhances secret symbol propagation by taking account extra implicit information flows, and constructs constraints to check ciphertext side channels (§4). During implementation, we put two "bounds" on the symbolic execution phase, i.e., the loop unroll parameter and the maximal analysis time of each assembly function. We set the loop unroll for two (more loop unrolling should not enable detecting more flaws), and the maximal processing time of each function is 30 minutes (see Table 2 for the number of timeout functions).

Compilation. The current implementation of CIPHERH analyzes 64-bit ELF binary compiled on x86-64 architectures. We use Clang (ver. 9.0.1), the LLVM C frontend, to compile cryptographic software into LLVM IR for taint analysis. We use the default compilation toolchain of the cryptographic software to compile cryptographic software into binary code for symbolic execution. At this step, we configure Clang with the same optimization level as in the cryptographic software's compilation configuration (usually `-O2` or `-O3`). When using the *same* optimization level, taint information in LLVM IR can be smoothly mapped to corresponding instructions in the assembly code. We do not observe cases where LLVM IR functions have different number of parameters in comparison to its assembly counterpart. Tainted callsites in LLVM IR can also be found in the assembly code easily using the callee name. We do not need to precisely match tainted memory load in LLVM IR with assembly code (which is difficult). As noted in §5.2.1, we assign a memory load from heap/globals/previous stack frameworks with a key symbol whenever any tainted data is loaded from memory during taint analysis.

7 Evaluation

We evaluate CIPHERH on ECDSA, RSA, and ECDH implementations of several real-world cryptographic libraries, including WolfSSL, OpenSSL and MbedTLS. To demonstrate the use of each cryptographic algorithm, we write a sample program for ECDSA signature and verification, RSA encryption and decryption, and ECDH key exchange, respectively. We compile all test cases into 64-bit ELF binaries on Ubuntu 18.04. We use the default optimization level for the evaluations. But we also analyze how optimizations may affect ciphertext side-channel vulnerabilities in §7.6.

Result Overview. We present the vulnerable program points detected by CIPHERH in Table 1. As reported, all the evaluated implementations contain memory writes that are vulnerable to ciphertext side-channel attacks. In sum, we find a total of 236 vulnerable program points from these evaluated cases, where 83 are from intra-procedural analysis and 153 are from inter-procedural analysis. We report the statistics of taint analysis in the last column of Table 1: for all evaluation settings, a considerable number of functions are covered during the taint analysis, where we taint in total 369 functions (about 13.6%). Moreover, we report the number of functions analyzed during (inter-/intra-) symbolic execution, and functions that contain ciphertext side-channel vulnerabilities in the "Vulnerable Function" columns. From a conservative point of view, it is reasonable to assume that developers may fix most leakages in a function at once. Thus, instead of "overclaiming" the findings using vulnerable program points, we view these 23 functions found by intra-procedural analysis and 25 functions found by inter-procedural analysis practically revealing vulnerabilities of common cryptographic libraries under ci-

Table 1: Evaluation results of different cryptographic algorithm implementations. **Opt.** denotes the optimization levels used when compiling executables.

Implementation	Algorithm	Opt.	Intraprocedural Symbolic Execution		Interprocedural Symbolic Execution		Function
			(Vulnerable/Analyzed) Functions	Vulnerable Program Points	(Vulnerable/Analyzed) Functions	Vulnerable Program Points	(Tainted/Covered)
WolfSSL 5.3.0	ECDSA	-O2	3/53	6	1/2	12	53/92
WolfSSL 5.3.0	RSA	-O2	3/30	14	3/5	30	30/78
OpenSSL 3.0.2	ECDSA	-O3	4/68	6	4/11	29	68/1061
OpenSSL 3.0.2	RSA	-O3	9/142	53	11/38	55	142/1296
MbedTLS 3.1.0	ECDH	-O2	2/37	2	2/5	5	37/87
MbedTLS 3.1.0	RSA	-O2	2/39	2	4/7	22	39/83
Total			23/369	83	25/68	153	369/ 2697

phertext side channels. Our follow-up analysis is primarily based on these vulnerable functions.

During taint analysis, OpenSSL shows many times the number of covered functions than that of WolfSSL and MbedTLS. Comparing to OpenSSL, as a fully-fledged cryptographic library, we find that WolfSSL and MbedTLS subsume relatively lightweight cryptographic implementations designed to usually low-cost embedded devices. Accordingly, we find that the OpenSSL case also contains a vast majority of the vulnerable program points (143 out of totalling 236) found by CipherH.

Existing research [33] has successfully exploited the ciphertext side channels and extract secret keys from the ECDSA implementation in OpenSSL that uses the constant time swap-based Montgomery ladder algorithm. We report that CipherH automatically discovered these findings. More importantly, CipherH discovers a substantial number of vulnerable program points in the RSA and ECDH implementations, which are *unknown* to the research community. We present case studies in §7.2–7.3. Also, it is known that WolfSSL has timely patched the ciphertext side-channel vulnerabilities exploited in [33]; two patches [57] are applied toward the `ecc_mulmod` and `mp_cond_swap_ct` functions in the ECDSA implementation. CipherH discovers zero vulnerabilities in `ecc_mulmod`, indicating the validity of the patch. Nevertheless, we report surprising findings that the other patch remains vulnerable, as reported by CipherH. The flaw in the patched code is presented in §7.2.1, whereas other confirmed findings are presented in §7.4. The developers of WolfSSL prepared new patches for our findings, which were re-analyzed by CipherH and revealed no vulnerabilities.

Table 2: Processing time of diff. cryptographic algorithm.

Implementation	Algorithm	Processing Time		Timeout Functions
		Taint Analysis	Symbolic Execution	
WolfSSL 5.3.0	ECDSA	118.3s	12300.1s	3
WolfSSL 5.3.0	RSA	108.7s	9660.6s	3
OpenSSL 3.0.2	ECDSA	467.1s	18960.2s	5
OpenSSL 3.0.2	RSA	465.3s	36720.4s	6
MbedTLS 3.1.0	ECDH	86.1s	10140.3s	2
MbedTLS 3.1.0	RSA	88.6s	10980.7s	2
Total		1334.1s	98762.3s	21

Processing Time. Table 2 reports the analysis time. Symbolic execution is more time consuming than taint analysis. CipherH completes taint analysis for *all* cases within 23 minutes. For most functions, performing symbolic execution takes less than ten minutes. However, due to path explosion, the analysis of a few functions did not finish within 30 minutes, which were manually terminated and labelled as timeout. The number of timeout functions is reported in the last column of Table 2. Overall, the results suggest that CipherH is efficient enough to analyze real-word cryptographic code.

Table 3: Efficiency comparison with prior relevant tools.

	Abacus	CacheS	CacheAudit
RSA/OpenSSL	failed (in a few seconds)	failed	failed
RSA/MbedTLS	failed ($\approx$ 7.3h)	failed	failed
ECDH/MbedTLS	timeout ($>$ 18h)	failed	failed

To compare analysis speed of CipherH with *cache* side channel detectors, we run three representative tools, Abacus [6], CacheS [51], and CacheAudit [19], to analyze RSA/OpenSSL, ECDH/MbedTLS, and RSA/MbedTLS. Abacus performs symbolic execution and constraint solving toward an execution trace (logged by Pin). It is optimized for speed and shows a significant improvement in efficiency compared to CacheD [52]. CacheAudit and CacheS perform static abstract interpretation [15] on the program call graphs. We compile our test cases into 32-bit x86 executables as they cannot analyze 64-bit executables. Table 3 reports the evaluation results. None of them are scalable to analyze our test cases.

Abacus fails in analyzing RSA/OpenSSL and RSA/MbedTLS. For ECDH/MbedTLS, it cannot finish symbolic execution on an execution trace after 18 hours. Abacus launches trace-based symbolic execution over a tainted trace, meaning that it explores less behaviors (and detects fewer bugs). Moreover, when analyzing production cryptosystems, there are often hundreds of caller/callee functions on an execution trace. It is seen that the symbolic formulas are accumulated and formula sizes are growing during analysis, making constraint solving slower and more costly. CipherH performs function-level, static symbolic execution toward each tainted function. The symbolic state at each function entry point is "empty" (including only unbounded symbols). Thus, CipherH is generally facing much simpler constraints, irrelevant of the number of involved functions. CipherH takes about three hours to analyze this case, and CipherH is more comprehensive than Abacus as it performs static symbolic execution.

CacheAudit and CacheS perform sound (or "soundy", in the language of CacheS) static abstract interpretation which is heavyweight. Their modeled symbolic, abstract domains, and cache status are generally complex. We also find that CacheAudit/CacheS fail to support some instructions in the test cases; adding support for these cases is challenging on our end, as it needs to define new abstract operators in their abstract domains, cache models, and prove soundness accordingly. More importantly, they need to first extract a subgraph (with a few caller/callee functions) to reduce the complexity. Thus, analyzing the entire call graph of production cryptographic code like OpenSSL and MbedTLS seems impractical.

We also note that though CipherH exhibits higher scalability in this comparison, the scope of several existing cache side channel detectors differs from that of CipherH. In particular, CacheAudit is a verifier not a detector, whereas both CacheAudit and Abacus perform quantitative analyses rather than qualitative ones like CipherH.

Code Release & Vulnerability Disclosure. We release CipherH at [1]. We have reported all findings to developers and discussed with them about patching the vulnerabilities. We have also provided developers with CipherH and detailed instructions for their early adoption, such that the developers could identify and fix other vulnerabilities by themselves. By the time of writing, we have received responses from both OpenSSL and WolfSSL. OpenSSL developers have confirmed our findings and are willing to work on software-level patches, which may take time and effort. WolfSSL developers have already issued a new patch to fix the vulnerabilities discovered on their old patch. Furthermore, another patch for the modular exponentiation algorithm has been provided (see §7.4). We anticipate that they will make announcements about the patched vulnerabilities.

7.1 Result Validation

CipherH has found in total 236 vulnerable program points, as reported in Table 1. In this section, we explore each finding and confirm if they are true positives. A true positive means that the ciphertext changes associated with the flagged program point indeed depend on the user-labelled secrets.

Intra-procedural Results. During intra-procedural symbolic execution, CipherH forms a constraint (Eq. 4) to check the existence of side-channel vulnerabilities. When Eq. 4 is satisfiable, the constraint solver will provide a pair of solutions (counterexamples) k_1, k_2 and k_1', k_2'. We opportunistically leverage the provided solutions to validate the test results. In particular, to valid each vulnerable program point, we instrument the vulnerable function's entry point and modify the secret bytes using either k_1, k_2 or k_1', k_2'. Note that these counterexamples should satisfy path constraints C covering two memory writes using secrets. We then compile the instrumented source code into two binaries. We execute both these binaries and monitor the ciphertext of the block corresponding to the vulnerable memory write. If a program point is a true positive, we expect to see that the ciphertext stays the same when using k_1, k_2 whereas it is changed when using k_1', k_2'. We performed such tests on all 83 findings in Table 1 and confirmed that they are all true positives.

Inter-procedural Results. We rely on specific patterns to detect ciphertext side-channel vulnerabilities related to reentrancy of a callee function F from a caller function F_c's loop statements (§5.2.2). Since no constraint solving is performed, we had to validate the results manual efforts. Specifically, we manually comprehend how F is invoked repetitively in F_c, as well as how secrets are passed via F's parameters and stored in memory. To ensure the accuracy of our findings, two of the authors examined and cross-checked all findings. Both authors have sufficient experience in side-channel attacks and cryptographic library analysis. We confirmed that 142 out of 153 findings are true positives.

The remaining 9 cases are false positives. These nine false positives are present in nine of 25 reported functions in Table 1. Figure 6 shows one of the false positive cases in RSA/MbedTLS. `nlimbs` is tainted and written to `X->n` at line 5. Since function `mbedtls_mpi_grow` is called in a loop in its caller function, our pattern-based inter-procedural analysis treats this function as vulnerable, and CipherH flags line 5 as "vulnerable". However, as the path constraint at line 3 ensures that `nlimbs` does not equal `X->n`, the memory write at line 5 always modifies `X->n`. Therefore, the ciphertext of the block containing `X->n` will continue to change whenever line 5 is executed, leaking zero information to the attacker. The other eight false positive cases exhibit similar patterns.

```
1. int mbedtls_mpi_grow(mbedtls_mpi *X, size_t nblimbs){
2.    ...                              // nlimbs is tainted
3.    if( X->n < nblimbs ) {   // path constraint
4.        ...
5.        X->n = nblimbs;   //the reported program point
6.        X->p = p;
7.    }
```

Figure 6: A false positive flagged by CipherH.

7.2 Case Study of ECDSA/ECDH

This section describes in detail the vulnerabilities found by CipherH in algorithms associated with elliptic curve (including ECDSA and ECDH).

7.2.1 ECDSA in WolfSSL

Figure 7 depicts the code snippet of a patch of WolfSSL (Ver 5.3.0) in response to the ciphertext side channels reported by Li et al. [33]. Function `ecc_mulmod` executes scalar multiplication based on Montgomery ladder, and the main loop at line 3 obtains each bit of the secret to perform *conditional swap* (line 6), *double* (line 11), and *add* (line 13). Compared to the unpatched version (see Appendix A), the countermeasures feature two components: ① Incrementing the variable `swap` (which reflects each bit of secret) at each iteration so that the same ciphertext is never observed; and ② copying the output of conditional swap to different memory addresses. CipherH reports no flaw of ①, since the constraint solver fails to find a satisfiable solution for the involved consecutive memory accesses. Nonetheless, for ②, CipherH discovers new vulnerable program points. The primary reason is because the *double and add* operations only modify one of the two swapped variables. If the unchanged variable is written back to its previous location, the same ciphertext can be observed.

Specifically, consider the *i*th iteration of the loop, in which `R[0]` and `R[1]` are copied to `R[2]` and `R[3]` according to

```
1. static int ecc_mulmod(const mp_int* k, ...) {
2.   ...
3.   for(i = 1,j = 0,cnt = 0; (err == MP_OKEY) && (i < t); i++) {
4.     ...
5.     swap += (kt->dp[j] >> cnt) + 2;  // obtain bits from exponent
6.     ecc_cond_swap_into_ct(R[(2-set)+0], R[(2-set)+1)],
7.             R[set+0],  R[set+1], modulus->used, swap);
8.     set = 2 - set;  // change to operate on set copied into
9.     // ensure 'swap' changes to a previously unseen value
10.    swap += (kt->dp[j] >> cnt) + swap;
11.    ecc_projective_dbl_point_safe(R[set+0],     // double points
12.                     R[set+0], a, modulus, mp);
13.    ecc_projective_add_point_safe(R[set+0],      // add points
14.            R[set+1], R[set+0], a, modulus, mp, &infinity) }
15.  ...
16. }
```
Figure 7: The elliptic curve scalar multiplication implementation from WolfSSL. Patches have been applied to defend against ciphertext side channels in the code snippet.

the value of $\mathtt{swap}_i$ ($\mathtt{swap}_i$ refers to the last bit in swap during the ith loop iteration) at line 6. Then, R[2] is doubled at line 11, and assigned the sum of R[2] and R[3] at line 13. Note that R[3] remains unchanged in the above operations. In the $(i+1)$th iteration, R[2] and R[3] are written back to R[0] and R[1] depending on $\mathtt{swap}_{i+1}$. To ease understanding, we list the possible value of R[0] and R[1] after the conditional swap of R[2] and R[3] according to different combinations of $\mathtt{swap}_i$ and $\mathtt{swap}_{i+1}$ in Table 4. An observation is that two of the four possible combinations of $\mathtt{swap}_i$ and $\mathtt{swap}_{i+1}$ can be leaked by monitoring the ciphertext of $\mathtt{swap}_i$ and $\mathtt{swap}_{i+1}$. Specifically, the same ciphertext of R[1] implies that both $\mathtt{swap}_i$ and $\mathtt{swap}_{i+1}$ are 0, while both equal 1 are reflected by the same ciphertext of R[0]. The remaining two cases share the same ciphertext change pattern (*i.e.*, both ciphertext of R[0] and R[1] change). As aforementioned, the WolfSSL developers confirmed this flaw and have prepared a new patch.

Table 4: The value in R[0] and R[1] after performing conditional swap twice with different combinations of $\mathtt{swap}_i$ and $\mathtt{swap}_{i+1}$. a and b represent the initial value of R[0] and R[1], respectively. The same ciphertext observed are boldfaced.

$\mathtt{swap}_{i+1}$ / $\mathtt{swap}_i$	0	1

0	R[0] = 2*a + b
	R[1] = b
		R[0] = b
		R[1] = 2*a + b

1	R[0] = 2*b + a
	R[1] = a
		R[0] = a
		R[1] = 2*b + a

Quantitative Analysis. Though CIPHERH is *not* designed to provide quantitative leakage analysis, we manually analyzed the above flaw from a quantitative standpoint. A uniformly distributed, k-bit secret has a total of 2^k possibilities. By exploiting the above flaw, once the same ciphertext of R[0] or R[1] are observed, the attacker can infer $\mathtt{swap}_i$ and $\mathtt{swap}_{i+1}$, then each bit of the secret can be recovered in sequence. An extreme case is that both R[0] and R[1] keep changing so that the possible values of the secret are reduced to two. Furthermore, considering other attack primitives [41] [3], even a minor leak of information from ECDSA scalar may likely result in full key compromise.

7.2.2 ECDH in MbedTLS

CIPHERH also reported vulnerabilities in MbedTLS's ECDH implementation. Four inter-procedural vulnerable program points are detected in function mbedtls_mpi_safe_cond_swap. We list its code snippet in Appendix B. These reported vulnerabilities are related to conditional swap operations. From the code snippet, it is seen that each bit of the secret is leaked via ciphertext side channels. Consistent with previous work [33], CIPHERH confirms the same vulnerable code snippet also exists in the ECDSA implementation of OpenSSL.

7.3 Case Study of RSA Vulnerabilities

CIPHERH successfully detects a number of vulnerable program points in the RSA implementations. Note that for the three evaluated RSA implementations, OpenSSL and MbedTLS use the fixed window-based modular exponentiation, while WolfSSL uses the montgomery ladder-based one. We elaborate on RSA/OpenSSL and RSA/MbedTLS in this section, and leave discussion about RSA/WolfSSL in §7.4.

7.3.1 RSA in OpenSSL

The fixed window-based modular exponentiation has a loop statement in which, for each loop iteration, a window is used to fetch several key bits and to query a pre-computed lookup table. Figure 8 depicts the table lookup function in OpenSSL. The parameter idx and buf of function MOD_EXP_CTIME_COPY_FROM_PREBUF denote the window-size key and the table base pointer, respectively. The inter-procedural analysis of CIPHERH finds a vulnerable program point at line 9, when one element of the pre-computed table is written to a BIGNUM b. Note that function MOD_EXP_CTIME_COPY_FROM_PREBUF is called iteratively in a loop statement within function BN_mod_exp_mont_consttime, causing memory writes at line 9 to be consecutive. Moreover, note that this denotes a strong dictionary attack opportunity, given that attackers can build the mapping between the plaintext and ciphertext of b using the dictionary attack to reveal window-size key idx. Due to compiler optimization, acc is deemed as safe, since its value is passed via a register rather than the stack in assembly (we use the -O3 optimization, the default configuration of OpenSSL); see discussion on how different compiler optimizations affect ciphertext side channels in §7.6.

Quantitative Analysis. For each window-size secret idx with L bits (L is usually 5 on 64-bit x86 platforms), there are 2^L possible values for idx. When L=5, for two secret segments, there is 1/32 probability that they are identical, which

```
static int MOD_EXP_CTIME_COPY_FROM_PREBUF(BIGNUM *b, int top,
                 unsigned char *buf, int idx, int window) {
  ...
  for (int i = 0; i < top; i++, table += width) {
    BN_ULONG acc = 0; // temporary variable storing bignumber
    for (j = 0; j < width; j++) {    // conditional assign
      acc |= table[j] & ((BN_ULONG)0 -
                     (constant_time_eq_int(j,idx)&1));}
    b->d[i] = acc;  }  // write the pre-computed value to b
  ...
}
```

Figure 8: The table lookup process in OpenSSL. OpenSSL uses the fixed window-based modular exponentiation, which needs to query a pre-computed table with several key bits.

directly gives attackers the knowledge whether two 5-bit key segments are identical or different. The security guarantee of RSA with *len(key)*-bit entropy then no longer holds.

7.3.2 RSA in MbedTLS

Fixed window-based modular exponentiation is also implemented in MbedTLS. We detected vulnerable functions sharing similar root cause with the OpenSSL case. Overall, when using the window-size key to query the lookup table, the query output is conditionally written to a BIGNUM. This memory write is vulnerable to ciphertext side channels. Specifically, we believe that the collision attack can be launched to determine whether the fetched value is expected, allowing the attacker to directly infer each window-size key and ultimately recover the entire secret key.

7.4 Advantage of Static Path Exploration

CipherH features a hybrid analysis, where we switch to traverse all paths of each tainted function using symbolic execution. During evaluation, we find several vulnerabilities on paths that are not covered during the dynamic taint analysis phase. This justifies the importance of launching static path exploration. In contrast, prior works like CacheD [52] only performs symbolic execution toward the tainted execution trace for the sake of scalability.

Figure 9 shows the code snippet of function `fp_exptmod` from RSA/WolfSSL, where the parameter `G` denotes the base and `X` is the secret exponent. Under the default compilation setting, function `_fp_exptmod_base_2` is inlined in function `fp_exptmod`, which enables the exploration of static symbolic reasoning. Note that when the base is 2, function `_fp_exptmod_base_2` , which is not covered during dynamic taint tracking, will be invoked instead of function `_fp_exptmod_ct` (covered during taint analysis).

Though `_fp_exptmod_base_2` is not executed by taint analysis, the static symbolic execution can cover it, and successfully reports two intra-procedural vulnerable program points at line 18 and line 20. Specifically, variable `buf` stores digits in secret `X`, and variable `y` indicates each bit of the secret.

```
1. int fp_exptmod(fp_int * G, fp_int * X, fp_int * P, fp_int * Y){
2.   ...
3.   if (X->sign == FP_NEG)
4.     ...
5.   else if (G->used == 1 && G->dp[0] == 2) {
6.     return _fp_exptmod_base_2(X, X->used, P, Y);}
7.   else {  // positive exponent so just exptmod
8.     return _fp_exptmod_ct(G, X, X->used, P, Y);}
9. }
10.
11. static int _fp_exptmod_base_2(fp_int * X, int digits, fp_int * P,
12.                     fp_int * Y) {  // is called when the base is 2
13.   ...
14.   for (;;) {
15.     buf    = X->dp[digidx--];
16.     // grab the next msb from the exponent
17.     y = (int)(buf >> (DIGIT_BIT - 1)) & 1;
18.     buf   <<= (fp_digit)1;
19.     // add bit to the window
20.     bitbuf |= (y << (WINSIZE - ++bitcpy));
21.     ... }
22. }
```

Figure 9: The code snippet of function `fp_exptmod` and function `_fp_exptmod_base_2` from WolfSSL. `_fp_exptmod_base_2` is called when the base is 2 to calculate 2^k. In this case, each bit of the exponent k can be leaked by ciphertext side channels.

To fill the window, `y` is added to `bitbuf`. However, an observation is that when `y` is equal to 0, the write operation at line 20 does not change the value of `bitbuf`. Hence, the collision attack can be launched to infer the corresponding `y` (whether it equals zero or not) during each write to `bitbuf`. As aforementioned, this flaw has been reported to the developers. They confirmed the finding and prepared a patch. We analyzed their patch using CipherH and found no vulnerability.

7.5 Pattern-Based Search

We use pattern-based search to identify inter-procedural vulnerable program points. The patterns compose ① a callee function F is repeatedly called from the same callsite in its caller, and ② F's parameters are tainted and written into memory. ① and ② are owing to observations over real-world cryptographic software. Here, we study whether and to what extent these patterns affect the presence of inter-procedural analysis findings. Given manual efforts are extensively involved at this step to confirm each finding, we use the ECDH/MbedTLS case for the study. The results are reported in Table 5.

Table 5: Inter-procedural findings for the ECDH/MbedTLS case by different patterns.

Pattern	Function (Vulnerable/Analyzed)	Inter-Procedural Vul. Program Points	False Positives
① & ②	2/5	5	1
①	8/19	17	10
Nil	11/35	21	14

CipherH yields more findings when ② is disabled (the 3rd row in Table 5). Specifically, 17 vulnerable program points are reported, residing within eight different functions. We manually studied these findings and identified ten false positives. Compared to the findings when enabling ②, CipherH reports 12 more program points, out of which nine are false positives and three are new true positives. Overall, we believe imposing ② as reasonable, given that it facilitates excluding a large number of false positive cases in exchange for a small number of false negatives.

The “Nil” setting in Table 5 indicates that both ① and ② are disabled. Holistically, this requires us to flag a pair of functions as “vulnerable” if they are consecutively called, and both write secrets to memory. Nevertheless, our preliminary study shows that this basic setup introduces too many false positives and is too obscure for manual investigation of each case. Therefore, we measure a simpler setting by flagging a vulnerable function F if it is called twice and writes secret to memory. We find that 11 out of 35 functions analyzed by CIPHERH are deemed vulnerable. Four more vulnerable program points are identified compared to the result of enforcing only ①. We manually studied these new reports, and found that they are *all* false positives: secret memory writes in F are toward distinct addresses due to different function inputs. In conclusion, we interpret that the two patterns ① and ② are adequate for delivering a scalable inter-procedural analysis with convincing accuracy and low false positive rates.

7.6 Compiler Optimization

In the preceding experiments, the default optimization settings (`-O2` for WolfSSL and MbedTLS, `-O3` for OpenSSL) were used. Nevertheless, it is obvious that the compiler optimization affect the vulnerabilities. Aggressive optimization tends to place variables into registers, resulting in less memory writes and thus less vulnerabilities, which is the case for RSA/WolfSSL as shown in Table 6.

Table 6: Intra-procedural findings for RSA/WolfSSL under different optimization levels.

Optimization Options	Functions Number (Vulnerable/Analyzed)	Intra-Procedural Vul. Program Points	Function Number (Tainted/Covered)
`-O2`	3/30	14	30/78
`-O0`	12/69	33	69/153

```
1. static int _fp_exptmod_ct(fp_int * G, fp_int * X, int
2.                      digits,  fp_int * P, fp_int * Y) {
3.    ...
4.    for (;;) {
5.      if (--bitcnt == 0) {
6.         ...
7.         // read next digit and reset bitcnt
8.         buf    = X->dp[digidx--];
9.         bitcnt = (int)DIGIT_BIT;  }
10.     // grab the next msb from the exponent
11.     y     = (int)(buf >> (DIGIT_BIT - 1)) & 1;
12.     buf <<= (fp_digit)1;
13.   ...  }
14.}
```

Figure 10: The code snippet shows that two more vulnerable program points are detected when using `-O0` for compilation.

Consider Figure 10, where function `_fp_exptmod_ct` executes the montgomery ladder-based modular exponentiation algorithm. During each loop iteration, one bit of the secret exponent is fetched at line 11. When WolfSSL is compiled using `-O0`, `y` and `buf` are stored on the memory stack, and memory writes to them introduce ciphertext side-channel vulnerabilities. Specifically, two vulnerable program points are detected at line 11 and line 12 when CIPHERH performs intra-procedural analysis. With these two vulnerabilities, attackers can infer the relations between every pair of adjacent bits, by observing the ciphertext changes in the memory block containing `y`. Given that `y` equals 0 or 1, the possible values of the secret are reduced to two. The other 17 newly-found flaws in WolfSSL (compiled using `-O0`) have similar causes.

8 Related Work

Static Analysis-Based Approaches. The majority of existing efforts in this field employ static analysis to detect side channels in cryptographic software implementations. CacheAudit [13, 19] uses abstract interpretation to model secret-dependent cache status and to quantify information leaks. CaSym [10] locates program points vulnerable to cache side channels using symbolic execution and constraint solving. Nevertheless, it has limited scalability and analyzes small programs. SymSC [25] evaluates timing side channels when considering thread interleaving. [49, 50] use type systems and program synthesis to detect and mitigate power side channels. Debreach [44] detects and mitigates compression side channels in PHP code with taint analysis and constraint solving. Their threat models are distinct with cache side channels. CacheD [52] and Abacus [6] also perform symbolic execution, but only over a single execution trace. In this manner, the cost of performing symbolic execution and constraint solving is substantially reduced, allowing for the analysis of production cryptographic software. Given the generally low scalability of abstract interpretation, CacheS [51] optimizes the abstract symbolic formula by only tracking coarse-grained non-secret information in cryptographic software. CacheS appears to report the best scalable findings of all static analysis tools. Nevertheless, it primarily analyzes a subgraph of the program call graph. We show that it still cannot perform whole-program static analysis for real-world cryptographic software in §7. In contrast, CIPHERH explores a *hybrid* analysis approach in which it uses dynamic taint analysis to track secret propagation throughout the whole cryptographic software, and it switches to static symbolic execution to analyze every path of each tainted function. Taint tracking offers a comprehensive, inter-procedural view of secret usage, and we limit expensive symbolic execution to each individual function. As discussed in §4, a promising advance, namely relational symbolic execution (RelSE) [21, 43], has also been employed for side channel analysis [16, 17]. We leave extending CIPHERH with RelSE as a future direction to enhance scalability.

Dynamic Analysis-Based Approaches. There also exist a number of dynamic approaches. DiffFuzz performs feedback-driven differential testing to detect a pair of inputs leading to timing side channels [42]. DATA [53, 54] and Stacco [58] apply trace differentiation analysis to decide secret leaks among a collection of execution traces. MicroWalk [56] quantifies and localizes side channel leakages over logged traces. Manifold [59] uses generative neural networks to synthesize private user images and text messages from traces. Existing works often require launching a large number of dynamic executions, and this is reasonable with conventional cache-based side channel leakage, in which attackers assess how different secrets lead to distinct cache access patterns or cache states. In ciphertext side channels, we however analyze how the same memory block is repeatedly accessed. This highlights the distinction between our work and previous dynamic methods.

9 Discussion and Limitations

False Negatives. CIPHERH is not sound, meaning that it may miss some leaks (i.e., CIPHERH has false negatives). On the one hand, functions that are not on the execution trace covered by taint analysis cannot be explored. On the other hand, some inter-procedural vulnerabilities can be neglected as we adopt pattern-based inter-procedural static analysis. The patterns we form are from observations of real cryptosystems and they are general, in the sense that we use the same patterns in our evaluation to analyze different cryptographic algorithms implemented in different cryptographic libraries. In §7.5, we experimentally demonstrated that the currently-formed patterns eliminate most false positives at the expense of a few false negatives. Moreover, pattern-based inter-procedural analysis is configurable, allowing developers to provide extra patterns for their domain-specific scenarios.

False Positives. CIPHERH may produce false positives, particularly in its inter-procedural analysis, as clarified in §5.2.2. Besides, due to the lack of context when analyzing a single function, our positive findings may be indeed unreachable when executing a program from the beginning. Nevertheless, our empirical results show that CIPHERH is very accurate with small number of false positives (9 out of 153).

Assembly Code Analysis. Cryptographic libraries may employ (inline) assembly code to offer architecture-specific implementations. We clarify that since CIPHERH uses LLVM DFSan to conduct taint analysis, the analysis will need to be launched on LLVM IR. In other words, we will have to configure the compilation toolchain to convert *source code* into LLVM IR for taint analysis, whereas the assembly language implementation of certain cryptographic libraries (e.g., the default modular exponentiation implementation of OpenSSL on x86 64-bit) is *not* analyzable.

Given that said, developers can manually mark secret variables for a specific assembly language function, and then use the symbolic execution module of CIPHERH to locate vulnerable program sites. We report that for the RSA assembly language implementation of OpenSSL, by manually annotating the taint information of the modular exponentiation functions on x86-64, we found vulnerabilities that are consistent with findings shown in Figure 8.

Proof of Concept (PoC) Exploits. CIPHERH flags assembly instructions vulnerable to ciphertext side channels. Similar to other tools in this field, CIPHERH cannot synthesize proof of concept (PoC) exploits. We believe it is exceedingly difficult for side channel detectors to "generate PoC attacks". Exploiting ciphertext side channels is a multi-step procedure [33] that requires pre-knowledge of the target cryptographic software and manual efforts. It is challenging to automate the exploitation process in an end-to-end manner (e.g., with a PoC script), let alone synthesize a PoC exploit using CIPHERH.

Padding Oracles. Oracle attacks were introduced by Bleichenbacher [8] to compromise PKCS#1 v1.5 encoding schemes. Multiple research works have shown the network security threat [7,31,39] due to Bleichenbacher attacks. While CIPHERH currently focus on secret leakage due to ciphertext side channels, its combination with oracle attacks may likely strengthen the attack surface and lead to even more severe outcomes. We leave it as one future work to explore.

10 Conclusion

CIPHERH is the first tool to systematically detect ciphertext side channels in production cryptosystems. It combines whole-program dynamic taint analysis and function-level static symbolic execution to deliver high scalability and comprehensiveness. We have applied CIPHERH to analyze cryptographic software and discovered a large number of flaws. Representative cases have been confirmed by the developers.

Acknowledgement

We thank the anonymous reviewers and our shepherd for their constructive feedback. We also thank developers of WolfSSL for their prompt responses and comments on our reported vulnerabilities. The authors from SUSTech are affiliated with both the Research Institute of Trustworthy Autonomous Systems and the Department of Computer Science and Engineering of SUSTech. This work is in part supported by SUSTech CSE-Ant Group Joint Research Platform on Secure and Trust Technologies.

References

[1] Research artifact. https://github.com/Sen-Deng/CipherH, 2022.

[2] AMD. Technical guidance for mitigating effects of ciphertext visibility under amd sev. https://www.amd.com/system/files/documents/221404394-a_security_wp_final.pdf, 2022.

[3] Diego F Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. Ladderleak:

Breaking ecdsa with less than one bit of nonce leakage. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 225–242, 2020.

[4] ARM. Arm Confidential Compute Architecture software stack. https://developer.arm.com/documentation/den0127/latest, 2021.

[5] UCSB & ASU. Angr. http://angr.io, 2018.

[6] Qinkun Bao, Zihao Wang, Xiaoting Li, James R Larus, and Dinghao Wu. Abacus: Precise side-channel analysis. ICSE, 2021.

[7] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient padding oracle attacks on cryptographic hardware. In Annual Cryptology Conference, pages 608–625. Springer, 2012.

[8] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In Annual International Cryptology Conference, pages 1–12. Springer, 1998.

[9] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 715–733, 2021.

[10] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In IEEE SP, 2018.

[11] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the 2008 USENIX Symposium on Operating Systems Design and Implementation (OSDI'08), 2008.

[12] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler. EXE:automatically generating inputs of death. In Proceedings of the 2006 ACM Conference on Computer and Communications Security (CCS'06), 2006.

[13] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. Quantifying information leakage in cache attacks via symbolic execution. TECS, 2019.

[14] Confidential Computing Consortium. Confidential Computing Consortium Members. https://confidentialcomputing.io/members/, 2022.

[15] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL. ACM, 1977.

[16] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. IEEE S&P, 2020.

[17] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse. In NDSS, 2021.

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In TACAS'08, 2008.

[19] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In USENIX Sec., 2013.

[20] Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. PLDI, 2017.

[21] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. Relational symbolic execution. In Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming, pages 1–14, 2019.

[22] gllvm. Sri-csl. https://github.com/SRI-CSL/gllvm, 2020.

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, PLDI '05.

[24] Michael Gruhn and Tilo Müller. On the practicability of cold boot attacks. In 2013 International Conference on Availability, Reliability and Security, pages 390–397. IEEE, 2013.

[25] Shengjian Guo, Meng Wu, and Chao Wang. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 377–388, 2018.

[26] Intel. Intel trust domain extensions whitepaper. https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-final9-17.pdf, 2020.

[27] Intel. Product brief, 3rd gen intel xeon scaleable processor for iot. https://www.intel.com/content/www/us/en/products/docs/processors/embedded/3rd-gen-xeon-scalable-iot-product-brief.html, 2021.

[28] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. Supporting Intel SGX on multi-socket platforms. 2021.

[29] David Kaplan. Upcoming x86 technologies for malicious hypervisor protection. https://static.sched.com/hosted_files/lsseu2019/65/SEV-SNP%20Slides%20Nov%201%202019.pdf, 2020.

[30] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. White paper, 2016.

[31] Vlastimil Klíma, Ondrej Pokornỳ, and Tomáš Rosa. Attacking rsa-based sessions in ssl/tls. In International Workshop on Cryptographic Hardware and Embedded Systems, pages 426–440. Springer, 2003.

[32] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In Srdjan Capkun and Franziska Roesner, editors, 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020, pages 487–504. USENIX Association, 2020.

[33] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A systematic look at ciphertext side channels on amd sev-snp. In 2022 IEEE Symposium on Security and Privacy (SP), pages 1541–1541. IEEE Computer Society, 2022.

[34] Mengyuan Li, Yinqian Zhang, and Yueqiang Cheng. CIPHERLEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In 30th USENIX Security Symposium (USENIX Security 21), pages 717–732, 2021.

[35] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CROSSLINE: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 2937–2950, 2021.

[36] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In 28th USENIX Security Symposium, pages 1257–1272, 2019.

[37] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In Annual Computer Security Applications Conference, 2021.

[38] LLVM. Dfsan. https://clang.llvm.org/docs/DataFlowSanitizer.html, 2020.

[39] James Manger. A chosen ciphertext attack on rsa optimal asymmetric encryption padding (oaep) as standardized in pkcs# 1 v2. 0. In Annual international cryptology conference, pages 230–238. Springer, 2001.

[40] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's virtual machine encryption. In 11th European Workshop on Systems Security. ACM, 2018.

[41] Phong Q Nguyen and Igor E Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. Designs, codes and cryptography, 30(2):201–217, 2003.

[42] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. Diffuzz: differential fuzzing for side-channel analysis. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 176–187. IEEE, 2019.

[43] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a doubt: testing for divergences between software versions. In Proceedings of the 38th International Conference on Software Engineering, pages 1181–1192, 2016.

[44] Brandon Paulsen, Chungha Sung, Peter AH Peterson, and Chao Wang. Debreach: Mitigating compression side channels via static analysis and transformation. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 899–911. IEEE, 2019.

[45] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. IEEE Journal on selected areas in communications, 21(1):5–19, 2003.

[46] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you C: Controlling side effects in mainstream C compilers. IEEE EuroS&P, 2018.

[47] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 21–41. Springer, 2012.

[48] Chungha Sung, Brandon Paulsen, and Chao Wang. CANAL: a cache timing analysis framework via LLVM transformation. ASE, 2018.

[49] Jingbo Wang, Chungha Sung, Mukund Raghothaman, and Chao Wang. Data-driven synthesis of provably sound side channel analyses. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 810–822. IEEE, 2021.

[50] Jingbo Wang, Chungha Sung, and Chao Wang. Mitigating power side channels during compilation. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 590–601, 2019.

[51] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying cache-based side channels through secret-augmented abstract interpretation. USENIX Security, 2019.

[52] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying cache-based timing channels in production software. In USENIX Security, 2017.

[53] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big numbers-big troubles: Systematically analyzing nonce leakage in (ec) dsa implementations. USENIX Security, 2020.

[54] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA – differential address trace analysis: Finding address-based side-channels in binaries. In USENIX Sec., 2018.

[55] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt of them all: Inference attacks against secure virtual enclaves. In ACM Asia Conference on Computer and Communications Security, pages 73–85. ACM, 2019.

[56] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A framework for finding side channels in binaries. In ACSAC, 2018.

[57] WolfSSL. Patches for ciphertext side channels. https://github.com/wolfSSL/wolfssl/pull/4666, 2021.

[58] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS'17. ACM, 2017.

[59] Yuanyuan Yuan, Qi Pang, and Shuai Wang. Automated side channel analysis of media software with manifold learning. USENIX Security'22.

A Unpatched ECDSA implementation in WolfSSL

```
static int ecc_mulmod(const mp_int* k, ecc_point* P, ecc_point* Q,...) {
  ...
  for (i = 1; (err == MP_OKAY) && (i < t); i++) {
    ...
    b = v & 1;   // v denotes the secret digits
    v >>= 1;
    // the value of swap is 0 or 1
    swap ^= b;
    if (err == MP_OKAY)  // swap R[0] and R[1] according to swap
      err = mp_cond_swap_ct(R[0]->x, R[1]->x, modulus->used, swap);
    if (err == MP_OKAY)
      err = mp_cond_swap_ct(R[0]->y, R[1]->y, modulus->used, swap);
    if (err == MP_OKAY)
      err = mp_cond_swap_ct(R[0]->z, R[1]->z, modulus->used, swap);
    swap = (int)b;
     // double and add points
    if (err == MP_OKAY)
      err = ecc_projective_dbl_point_safe(R[0], R[0], a, modulus, mp);
    if (err == MP_OKAY) {
      err = ecc_projective_add_point_safe(R[0], R[1], R[0], a, modulus,
                                                        mp, &infinity);
  }
  ...
```

Figure 11: The elliptic curve scalar multiplication implementation from WolfSSL without defining WC_PROTECT_ECVRYPTED_MEM. The attacker can infer secret k by ① monitor the ciphertext of swap. ② monitor the ciphertext of conditional swap in mp_cond_swap_ct.

B Vulnerabilities in ECDH/MbedTLS

```
1. int mbedtls_mpi_safe_cond_swap( mbedtls_mpi *X,  mbedtls_mpi *Y,
2.                                                   unsigned char swap) {
3.    ...    // swap reflects one bit of the secret
4.    limb_mask = mbedtls_ct_mpi_uint_mask( swap );
5.    s = X->s;
6.    // conditional swap of X->s and Y-> s
7.    X->s = mbedtls_ct_cond_select_sign( swap, Y->s, X->s );
8.    Y->s = mbedtls_ct_cond_select_sign( swap, s, Y->s );
9.    // conditional swap of X->p and Y->p
10.   for( i = 0; i < X->n; i++ ) {
11.      tmp = X->p[i];
12.      X->p[i] = ( X->p[i] & ~limb_mask ) | ( Y->p[i] & limb_mask );
13.      Y->p[i] = ( Y->p[i] & ~limb_mask ) | (     tmp & limb_mask );
14.    }
15.   ...
```

Figure 12: Conditional swap operations in ECDH implementation of MbedTLS. Four inter-procedural vulnerable program points are reported by CIPHERH at line 7, 8, 12 as well as 13. The attacker can monitor the ciphertexts of X and Y to infer swap.