An Empirical Study Measuring In-The-Wild Cryptographic Microarchitectural Side-Channel Patches

Sen Deng
The Hong Kong University of Science and Technology
Hong Kong, China
sdengan@cse.ust.hk

Zhibo Liu*
The Hong Kong University of Science and Technology
Hong Kong, China
zhiboliu@ust.hk

Shuai Wang
The Hong Kong University of Science and Technology
Hong Kong, China
shuaiw@cse.ust.hk

Yinqian Zhang*
Southern University of Science and Technology
Shenzhen, China
yinqianz@acm.org

Abstract

Patching microarchitectural side channels in real-world cryptographic software is a challenging task that does not always result in efficient and secure patches. Despite the continuous efforts of researchers and developers, the security and performance of microarchitectural side-channel patches have not been comprehensively studied before. To systematically study this patching effort, this paper conducts the first measurement study on in-the-wild side-channel patches, yielding the SideBench dataset comprising 165 patches from three mainstream cryptographic libraries (OpenSSL, WolfSSL, and MbedTLS), and offering an automated analysis tool, SideEval, tailored to analyze side-channel patches through a combination of dynamic taint analysis and static symbolic execution. Our analysis reveals that even among patches written by experienced developers, 25 are *insecure*, leaving residual side-channel leakages potentially unnoticed by developers for years. Furthermore, some patches rashly issued to fix one microarchitectural side channel may inadvertently open new leakages against other side-channel models. We also observed that patches in different cryptographic libraries, even when fixing the same code pattern, can incur drastically different overheads, varying from 10% to 170%. Additionally, our measurements show that recent rule-based and large language model (LLM)-based automated patching tools are not as secure as expected. We summarize our findings and provide insights for developers to fix side channels securely and efficiently.

CCS Concepts

• **Security and privacy → Side-channel analysis and countermeasures**;

Keywords

Cache side channel, Ciphertext side channel, Cryptographic libraries, Side-channel patches

ACM Reference Format:
Sen Deng, Zhibo Liu, Shuai Wang, and Yinqian Zhang. 2025. An Empirical Study Measuring In-The-Wild Cryptographic Microarchitectural Side-Channel Patches. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744881

1 Introduction

A large number of side channels have been discovered and exploited to defeat modern cryptographic schemes [11, 12, 15, 18, 19, 30]. Among these side-channel attacks, microarchitectural side channels are particularly hazardous and pervasive [38], where the adversary exploits the hardware features, e.g., CPU caches and Simultaneous Multithreading (SMT), to violate the system's security guarantee and steal secrets from cryptographic software [5, 6, 8, 21, 22, 35, 57, 67]. Disabling those hardware features to mitigate the risk of side channels is plainly impractical and can result in significant performance degradation. As a result, effective software patching of side-channel vulnerabilities has been an enduring objective.

Developers of cryptographic libraries have been patching their products promptly to prevent microarchitectural side-channel attacks. For instance, the OpenSSL project has been issuing patches to fix side-channel vulnerabilities since 2005.[1] Typically, a side-channel patch is created to prevent adversaries from exploiting a specific side-channel leakage (e.g., a secret-dependent branch). It modifies the program source code into a more secure version, e.g., converting a secret-dependent branch into a constant-time branch, or converting a secret-dependent table lookup into a constant-time version. Over the past two decades, many patches have been issued and deployed in three production cryptographic libraries (OpenSSL, WolfSSL, and MbedTLS). All these patches were created by experienced cryptographic software developers and went through rigorous code review and testing before being deployed.

Despite the efforts, the community lacks a systematic and in-depth measurement of those in-the-wild patches' quality, i.e., the degree to which a patch is *secure* and *efficient*. From the security perspective, a patch is secure if it can prevent adversaries from

*Corresponding authors.

CCS '25, Taipei, Taiwan, China.

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3744881

[1] To the best of our knowledge, the first patch is commit-0ebfcc8.

exploiting microarchitectural side-channel leakages. However, side-channel exploitations are often subtle to mitigate, where a patch may not comprehensively cover all leakage sites (e.g., all secret-dependent memory accesses in a vulnerable code snippet), and, unsurprisingly, an ill-implemented patch may introduce new side-channel leakages (see Sec. 3). We believe this situation will only get worse as the community increasingly discovers microarchitectural side channels under different scenarios [6, 16, 21, 22, 35, 67]. Without a thorough understanding of various microarchitectural side channels, developers will likely write patches that only mitigate partial leakages or even introduce new ones via different side channels. Performance is another critical aspect, given that cryptographic software is often used in performance-critical scenarios, e.g., secure data transmission. Nevertheless, we find multiple ways to write a legitimate patch toward one vulnerability, where overhead incurred by these patches varies largely, e.g., from 10% to 170%.

This paper fills this gap by conducting the first systematic measurement study on microarchitectural side-channel patches. Informally, we consider microarchitectural side channels as those that leak information through a program's memory access patterns, given the low attack requirement and large channel capacity for monitoring memory access patterns [38]. By cautiously searching from the GitHub repositories and version update logs, we form the first dataset, SideBench, which includes 165 side-channel patches issued in the past two decades from three mainstream cryptographic libraries, OpenSSL, WolfSSL, and MbedTLS. To our best knowledge, this dataset is the most comprehensive one to date, covering **all** side-channel patches issued in production cryptographic libraries.

Manually assessing these patches is hardly feasible due to the large number of patches and, more importantly, the high complexity of microarchitectural side channels. Thus, we developed SideEval, an automated tool tailored to assess side-channel patches on security and cost. SideEval conducts a series of dynamic taint analysis, static taint analysis, and symbolic execution to rigorously reason a given patch using secret information flow and two microarchitectural side-channel models (cache and ciphertext [17, 35]). SideEval also employs symbolic execution and a practical cost model to estimate the overhead incurred by a patch. To address the scalability issue in analyzing in-the-wild cryptographic software (which is often highly complex), SideEval uses three heuristics to trim call graphs in the context of a patch before applying rigorous albeit expensive symbolic execution and constraint solving.

Our study shows insightful findings. While all SideBench patches are written by experienced developers, we found that (1) 25 side-channel patches (10 in OpenSSL, 11 in WolfSSL, and 4 in MbedTLS) are insecure, leading to residual side channels neglected for years, and our study pushes for two new fixes by the time of writing; (2) 13 patches (7 in OpenSSL, 3 in WolfSSL, and 3 in MbedTLS) appear to fix the cache side channels, yet opening new side-channel leakages against ciphertext side channels; and (3) patches in different libraries, even issued to fix the same code pattern, can incur drastically different overhead (varying from 10% to 170%). We also leverage SideEval to measure three recent rule-based and large language model (LLM)-based automated patching tools [13, 59] and find that they are not as secure as stated (see Sec. 7.4). In sum, we make the following contributions:

- In line with the recent arms race between microarchitectural side-channel attacks and mitigation, we initiate a new focus to systematically measure side-channel patches from in-the-wild cryptographic software. We create the first dataset, SideBench, which includes **all** (165 in total) side-channel patches issued towards three mainstream cryptographic libraries.
- To deliver a systematic and efficient analysis, we introduce an automated tool, SideEval. SideEval offers a comprehensive security and performance measurement, and it also features a set of optimizations to improve analysis scalability. SideEval can be integrated into the standard CI/CD pipeline of cryptographic libraries to assist developers in daily development tasks.
- Our study indicates that those real-world patches may still contain severe security and performance issues, and recent automated patching tools are not as secure as expected. We hope our findings can raise awareness among developers and researchers to improve the quality of side-channel patches.

Artifact & Extended Version. The code, data, and an extended version of this paper (which contains additional results and the details of scalability optimization) are provided at https://github.com/SenDeng/SideEval [1].

2 Preliminary

2.1 Microarchitectural Side-Channel Attacks

Side-channel attacks fall into two main categories: microarchitectural side-channel attacks and physical side-channel attacks. Microarchitectural attacks enable adversaries to infer secret information by exploiting hardware characteristics, typically without requiring physical access to the target system. In contrast, physical attacks necessitate the attacker's proximity to measure physical emanations during execution, such as power consumption [18], electromagnetic radiation [19], or acoustic emissions [15]. Numerous microarchitectural side-channel attacks targeting cryptographic software have been discovered [6, 21, 22, 35, 67].

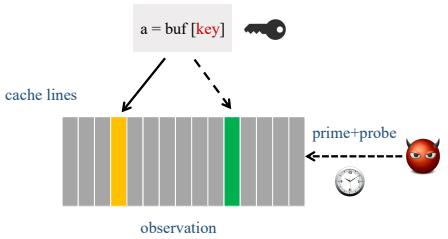
Focusing on the microarchitectural category, our preliminary investigation into three mainstream, widely-used cryptographic libraries (OpenSSL, WolfSSL, and MbedTLS)[2] reveals that developers frequently patch vulnerabilities related to memory access (specifically, cache and ciphertext side channels). However, vulnerabilities associated with other microarchitectural attack vectors, such as Spectre [29], often remain unpatched due to maintenance burdens [44]. Below, we briefly introduce cache side channels and ciphertext side channels with concrete attack examples; patches of these two will be our primary targets for this measurement study. We defer discussion of other side channels to Sec. 8.

Cache Side Channels. The shared cache units in the processor cause cache side channels. In short, cache side channels have been demonstrated as practical under different scenarios and adversarial models, such as access-based attacks [63], trace-based attacks [20], and eviction-based attacks [64]. These attacks can be mounted on a wide range of software like cryptographic libraries and web servers [35, 68]. We illustrate a cache side-channel attack in Fig. 1(a): a secret-dependent memory access results in a consequent cache line access that depends on the secret. Thus, the

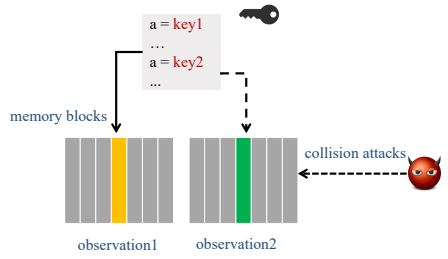
[2]These cryptographic libraries have nearly daily commits, dedicated maintenance teams, and frequent (typically every 2–3 months) releases.

attacker can infer the secret by observing the accessed cache access of the victim process using standard cache side-channel attacks like Flush+Reload [63] and Prime+Probe [45].

Nearly all studied cache side-channel vulnerabilities [14, 17, 52, 64, 65] and their patches in in-the-wild cryptographic libraries are derived from cache line/bank-based threats. Thus, SideEval considers both cache line and cache bank side-channel models, i.e., we assume that attackers can effectively observe the accessed cache line and/or cache bank of the victim process. This way, attackers can infer the secret information by analyzing the cache line/bank access pattern. We detail the side channel modeling in Sec. 5. We discuss other side channel modeling in Sec. 8.



(a) a sample cache side-channel attack.



(b) a sample ciphertext side-channel attack.

Figure 1: A schematic view of microarchitectural side-channel attacks. An attacker can infer the secret key by monitoring the cache lines (a) or the memory blocks (b).

Ciphertext Side Channels. Trusted Execution Environments (TEEs) are hardware extensions available in modern processors that can provide users' programs with integrity and confidentiality protection. The ciphertext side-channel attack is an attack where the attacker breaks the confidentiality of *deterministic encryption*-based TEEs by exploiting the information leakage caused by ciphertext changes in the encrypted memory. Ciphertext side channels were initially discovered on AMD SEV [35, 36], where hypervisors have read access to the encrypted memory's ciphertext, and can be extended to other deterministic encryption-based TEEs, such as Intel TDX [26], Intel SGX on Ice Lake SP [26, 28], and ARM CCA [9] by memory bus snooping [32]. Hardware vendors, such as AMD, have suggested addressing ciphertext side-channel vulnerabilities via software patching rather than resolving them directly in the hardware, due to the rather high overhead [7].

Ciphertext side-channel attacks can be classified into *dictionary attacks* and *collision attacks* [35]. Dictionary attacks are attacks where the adversary collects sufficient ciphertext-plaintext pairs at a fixed physical address and builds a dictionary, which is 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. Fig. 1(b) presents a schematic view of launching the collision attack, where `key1` and `key2`, 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 infer the relations (equality/inequality) of key bits, thus largely reducing the search space of the secret key.

In line with patches issued in in-the-wild cryptosystems that mitigate collision attacks, SideEval models collision attacks, indicating a more general and severe variant of ciphertext side-channel threats. We detail the side channel modeling in Sec. 5.

2.2 Side-Channel Patch Paradigms

This section reviews side-channel patch paradigms. To our observations, patches implemented in de facto cryptographic libraries are all instantiated from the paradigms reviewed in this paper.

Constant-Time. This paradigm is widely adopted to patch side-channel vulnerabilities. The idea is to convert secret-dependent branches into constant-time branches. For instance, Fig. 2 shows the procedure of a constant-time swap. Depending on the value of a secret decision bit `k`, the values `p` and `q` are swapped (`k = 1`), or left as-is (`k = 0`). Such constant-time conversion prevents attackers from inferring the secret by observing time variability and inconsistencies that result from execution of different secret keys, thereby mitigating the side channels.

Masking. The paradigm masks secret-dependent values using random values. For instance, a secret variable `k` can be masked with a random value `r` through a xor operation, as shown in Fig. 2 (denoted as `^`). Instead of eliminating the secret-dependent execution (as the constant-time paradigm does), this paradigm makes the recovered secret value meaningless to the attacker.

Blinding. This paradigm is also pervasively used. The idea is to blind secret-dependent values with random values. For instance, the message `m` to encrypt can be blinded with a random value `r` to get the blinded message `m'`, as shown in Fig. 2. This paradigm is similar to masking. Masking hides the data (i.e.,key) itself, while blinding randomizes the value processed by the secret operation. Therefore, it is more effective in resisting side-channel attacks on asymmetric cryptosystems like RSA [41], which often involve modular exponentiation or similar mathematical structures with a secret exponent/scalar.

Other Paradigms. There are other paradigms to patch side-channel vulnerabilities. For instance, AMD has recommended some mitigation paradigms against ciphertext side channels, including data moving, data padding, and data in register [7]. Examples of these code patterns will be analyzed in Sec. 7.

3 Motivation

Security. The security of a side-channel patch is the degree to which it can prevent adversaries from exploiting side-channel leakages. Ideally, a patch should comprehensively cover all leakage sites of a vulnerable code (e.g., the Montgomery ladder computation of RSA) without introducing new side-channel leakages. Nevertheless, our preliminary study revealed that writing a comprehensive patch is challenging, even for experienced developers. Therefore, a

```
cons_swap (p, q, k) : // constant-time paradigm:
     t = k & (p ^ q);   // bit k = 0 or 1
     p ^= t, q ^= t;

masking (k):  // masking paradigm:
     r = random(); // generate random value r
     a = k ^ r;

blinding (k): // blinding paradigm:
     r = random();
     m' = m * r^k;// m is the message to encrypt
```

Figure 2: Three primary side-channel patch paradigms.

patch may cover only partial leakage sites, and an ill-implemented patch may require repeated patches to fix the residual issues. For example, in WolfSSL's scalar multiplication function `ecc_mulmod`, we identified five repeated commits for side-channel vulnerabilities since 2015, whose details are shown in Table 1.

Table 1: Repeated patches against side channels in WolfSSL. ✕ denotes that the patch has cache/ciphertext side channels.

Commit	Cache Side Channels		Ciphertext Side Channels	Time
	Control Transfer	Memory Access		
Original Ver.	✕			
9d4fb79		✕		2015-01-08
46a0ee8		✕		2016-09-14
f0db2c1			✕	2020-10-16
0b2b218			✕	2021-12-16
0bcd38f				2022-06-06

The original version of function `ecc_mulmod` exhibits secret-dependent control transfers. Developers pushed a commit (`9d4fb79`) to eliminate the control transfers, but inadvertently created secret-dependent memory access. One year later, another update was committed to the same function, but the implementation remained incomplete.[3] It was not until 2020 that the cache side channel was finally eradicated. Regrettably, this also brought the ciphertext side channels (which did not exist in the unpatched code). The developers attempted to patch the vulnerability in 2021,[4] but it was not until 2022 that the ciphertext side channel was completely resolved. Overall, developers take seven years to ultimately fix this side-channel weakness. This shows that the side-channel vulnerability is usually concealed; developers need to consider a variety of patterns and different side-channel types to identify such vulnerabilities. Hence, developers face significant challenges in manually assessing the security of a patch. This observation motivates us to measure real-world patches in production cryptographic libraries and to evaluate their security comprehensively. We also develop an automated tool (SideEval) that can be integrated into the developers' workflow to assist in patching side channels regularly.

[3]Commit `46a0ee8` still contains secret-dependent memory access, leaving it vulnerable to cache side-channel attacks.
[4]Commit `0b2b218` incorrectly uses a "data moving" paradigm, which allows ciphertext patterns to remain observable.

Performance. Our preliminary study also found that performance is highly concerned by cryptographic software developers. We observed that in-the-wild cryptographic libraries are often performance-optimized from algorithmic and implementation perspectives. Production libraries like OpenSSL have been patched frequently to achieve better runtime efficiency.[5]

However, there are often multiple ways to patch a given side-channel vulnerability (we reviewed common paradigms in Sec. 2.2). We even found that developers may use different patch paradigms for different leakage sites within one patch commit. For example, in commit `0bcd38f` of WolfSSL, developers simultaneously fixed two ciphertext side-channel vulnerabilities in functions `_fp_exptmod_base_2` and `ecc_mulmod`. In line 6 of Fig. 3, developers use data masking to obfuscate the attacker's observations while moving secret data to different memory blocks in line 15. This observation leads to an important yet overlooked question: To what extent would side-channel patches impact efficiency while ensuring security, and if so, which patching implementations perform better in a given scenario? This question motivates us to measure the performance of side-channel patches.

```
1.  static _fp_exptmod_base_2(fp_int * X, int digits, ...){
2.      ...
3.      y = (int)(buf >> (DIGIT_BIT - 1)) & 1;
4.      buf   <<= (fp_digit)1;
5.      // Ensure value changes using WINMASK
6.      bitbuf += (WINMASK + 1) + (y << (WINSIZE - ++bitcpy));
7.      ...
8.      err = fp_mul_2d(res, bitbuf & WINMASK, res);
9.      ...
10. }
11.
12. static int ecc_mulmod(const mp_int* k, ecc_point* P, ...)_
13.     ...
14.     // Move secret data R[0] and R[1] into different location
15.     ecc_cond_swap_into_ct(R[(2-set)+0], R[(2-set)+1)],
16.                   R[set+0],  R[set+1], modulus->used, swap);
17.     set = 2 - set;
18.     err = ecc_projective_dbl_point_safe(R[set + 0], R[set + 0], a,
19.                                         modulus, mp);
20.         ...
21. }
```

Figure 3: Patch in WolfSSL commit `0bcd38f`. Masking and Moving paradigms are simultaneously used.

4 Study Overview

Table 2: Patch dataset collection. Opt. denotes the optimization levels used when compiling these libraries.

Implementation	Opt.	Commit History	Obtained Patches
OpenSSL	-O3	Since May 26, 2005	74
WolfSSL	-O2	Since Jan. 8, 2015	34
MbedTLS	-O2	Since Nov. 14, 2012	57
Total			165

SideBench Collection. We report the patch datasets used in this study in Table 2. We collect **all** side-channel patches issued over three mainstream cryptographic libraries—WolfSSL, OpenSSL, and

[5]For example, commit `9d91530` in OpenSSL leverages a specialized Montgomery ladder for binary curves to provide a specialized differential addition-and-double implementation. This speedups prime curves.

MbedTLS.[6] To do so, we inspect their repositories on GitHub to find side-channel patches issued since their first check-in (e.g., OpenSSL was first checked in on 2005-05-26) until 2025. We iterate each check-in commit and search for keywords like "side channel" and "timing attack" in the commit messages. We also manually check the patches and their corresponding commit messages to ensure they are indeed microarchitectural side channel fixes. For example, we screened out OpenSSL commit `848113a` and MbedTLS `CVE-2019-16910`, since they mitigate the "one-and-done" side-channel attack and multi-message sign side channel attack, respectively, which are out of the scope of this study. At this step, we prepare over 20 keywords to search for the patches (See Table 3). In total, we collect 165 patches from the three cryptographic libraries.

Table 3: The used keywords to search for related patches.

Search Term	Search Term Variations
side channel	side-channel
constant time	constant-time
timing attack	cache attack
microarchitectural attack	ciphertext attack
encrypted memory	memory block
cache line	cache bank
data masking	data padding
date moving	data in register
blinding	cache resistant
secret-dependent branch	secret-dependent memory access

We primarily study patches in cryptosystems compiled and executed on x86-64 platforms; all cryptosystems are written in C/C++ and compiled with GCC, a common setting where side-channel attacks are frequently reported and mitigated. Thus, to run these patches, we write a sample program to trigger each patch, such as ECDSA signature and verification, RSA encryption and decryption, and ECDH key exchange. We compile all test cases into 64-bit x86 ELF binaries on Ubuntu 18.04. We use the default optimization level for the evaluations (noted in Table 2) to emulate the common usage of these libraries. Nevertheless, our study methodology and SideEval itself can analyze patches in cryptosystems developed and executed in other settings. See extensibility discussions in Sec. 8.

Caveat. SideBench should have covered **all** microarchitectural side-channel vulnerabilities in those three popular libraries. Yet, it may not be inaccurate to assume that some patches may be missed due to the limitations of our data collection strategy. However, we believe SideBench is representative enough for our study, as the missed patches are likely rare and do not affect our findings and conclusions. Furthermore, the overall study methodology and the developed automated tool (SideEval) have provided a robust framework for evaluating future side-channel patches.

Attributes. This study focuses on analyzing the security and performance of microarchitectural side-channel patches. We target microarchitectural side channels related to a program's memory access patterns, i.e., cache side channels and ciphertext side channels. Importantly, our study (and SideEval) can be easily extended to analyze other side channels. Audiences may argue that ciphertext side channels exist only in TEE environments, while the cache side channels are present in nearly all contemporary cloud environments. To clarify, we deem a cryptographic library as "secure" if it is free from microarchitectural side channels in all mainstream scenarios; note that TEEs have been widely adopted in cloud environments and other commercial applications (e.g., mobile devices). As a proof, we see that the developers of cryptographic libraries like WolfSSL have been actively patching both ciphertext and cache side channels on their codebase. However, properly analyzing both attributes is uneasy; see our technical solutions in Sec. 5.

Automated vs. Manual Measurement. The inherent challenges of manual side-channel analysis, such as handling implicit secret flows and navigating complex codebases, have long motivated the development of automated techniques [17]. These difficulties become especially apparent when assessing the security and performance of side-channel patches. From the security aspect, developers may lack a thorough understanding of various microarchitectural side channels. Even experienced developers may write patches that only mitigate partial leakages or introduce new ones via different side channels. And from the performance aspect, manually measuring the overhead is also challenging, if not impossible.

We thus develop SideEval, an automated tool tailored to analyze side-channel patches; details are in Sec. 5. Besides measuring patches from SideBench, SideEval can be integrated into the development pipeline of cryptographic libraries.[7] Furthermore, SideEval is not specific to open-source projects; proprietary software developers can also employ SideEval in their daily development/-maintenance routines. Before issuing new patches, SideEval serves as a "patch debugger" to assist developers in assessing their patches regarding security and performance. SideEval is fully automated; developers can configure SideEval with the patch information, and SideEval conducts systematic analysis and reports potential issues. Developers can accordingly fix SideEval's findings and refine their patches before release.

Novelty Clarification. There are some existing tools that can analyze side-channel vulnerabilities in cryptographic software (reviewed in Sec. 9), although SideEval is the first to analyze side channel patches. We present empirical comparison between SideEval and existing tools in Sec. 7.4. Yet, we admit that SideEval and past tools share essential techniques like taint analysis and symbolic reasoning. Overall, developing a novel tool is *not* our main contribution. Instead, the main contribution is the first systematic measurement on in-the-wild side-channel patches. Nevertheless, we take credits for SideEval's tailored design to speedily and comprehensively analyze side-channel patches. As shown in Sec. 7.4, none of existing tools can satisfactorily analyze side-channel patches.

Moreover, we clarify that SideEval is *not* an attack tool; the exploitability of its findings (e.g., whether RSA private keys can be reconstructed via SideEval's findings) is beyond the scope.

5 SideEval Development

We design SideEval, an automated tool for benchmarking patches in SideBench. SideEval conducts a hybrid analysis pipeline to

[6] We initially selected six of the most popular cryptographic libraries on GitHub (based on star count): OpenSSL, LibreSSL, WolfSSL, MbedTLS, Libsodium, and Crypto++. Using keyword searches (see Table 3 for details), we narrowed our focus to the three libraries with the most side-channel patches. By the time of writing, the remaining libraries had much fewer side-channel patches (LibreSSL had 1, Libsodium had 6, and Crypto++ had 5). While this study centers on these three selected libraries, our methodology (along with our tool, SideEval) is designed to be broadly applicable to other cryptographic libraries as well.

[7] The WolfSSL team has already requested SideEval for integration and usage.

assess a patch's security and performance cost, offering different analysis modes to accommodate various scenarios. As depicted in Fig. 4, SideEval begins with scalable dynamic taint analysis to identify all secret-related variables within a patched function. It then switches to static taint analysis to analyze the validity of secret information flow over the patched code. Finally, SideEval employs rigorous static symbolic execution to analyze the patched code against microarchitectural side-channel models and quantify the introduced performance overhead. We provide a detailed discussion of security/cost modeling and scalability optimizations in Sec. 6.

Analysis Units. SideEval conducts analysis on a per-patch basis. A patch often involves changes in multiple statements. Thus, we take a function as the analysis unit, and we analyze a patch by analyzing a sub-call graph of the patched function and its callees. With the patch commit information, it is trivial to locate the patched function in the codebase for analysis. Knowledgeable audiences may argue that a security patch may involve changes in multiple functions [40]. However, according to our observation, side channel patches in cryptographic libraries rarely involve changes in multiple functions (all patches in SideBench change only one function). Instead, they often involve changing one and introducing new callees or removing existing callees. This way, we believe our definition of analysis unit is reasonable and practical.

Dynamic Taint Tracking. Once the patched function is located, the next step is to flag all secret inputs in the patched function. Here, we define the secret inputs as the function's parameters, the return values of its call sites, or memory load outputs dependent on the secret keys. To properly decide if any of its input depends on the secret, SideEval conducts dynamic taint analysis. SideEval's dynamic taint analysis relies on DataFlowSanitizer (DFSan) [37], an LLVM-based tool that tracks information propagation across variables and memory regions using comprehensive propagation rules. We configure the taint analysis to treat secret keys in the evaluated cryptographic software as taint source and instrument the patched function's entry point, memory loads, and call sites as taint sink points. This allows the analysis to identify flows from secret sources to these sink points. To run the evaluated cryptographic software, we either use built-in test programs (shipped with the analyzed libraries) or write simple test cases as needed.

Static Symbolic Execution. The symbolic execution module of SideEval is built over angr [3], a popular symbolic execution engine for binary code. All tainted variables derived from dynamic taint analysis will be symbolized as fresh key symbol k_i for symbolic execution. We perform symbolic execution on the patched function at the binary level, during which the constraint solver Z3 [43] is used to check the satisfiability of each path constraint. We further improve the symbolic execution process by automatically configuring the inter-/intra-procedural analysis settings according to the types of patches (see details in Section 6). The branch condition constraints and memory address constraints collected during symbolic execution will later be used to verify compliance with side-channel models in Sec. 6 and identify side-channel vulnerabilities. For performance evaluation, we select top-K paths with the highest number of tainted instructions to evaluate the overhead introduced by the patch. For scalability reasons, we set the loop unrolling at most twice. The evaluation platform is equipped with Intel core i7-12700 and 32 GB memory. The OS version is Ubuntu 18.04.

6 Modeling

6.1 Security Modeling and Analysis

To assess a patch's security guarantee, SideEval considers both the validity of secret information flow and several side-channel leakage models. We now present the details of these models.

Secret Information Flow Analysis. Inspired by traditional security information flow analysis [23], SideEval evaluates the patch's security by analyzing its secret information flow. Given an unpatched function F and its patched version F', SideEval analyzes the sub-call graphs of F and its callee functions $\mathbb{G}$, identifying a set of taint sink statements $\mathcal{S}$. Also, F' and its callee functions $\mathbb{G}'$ are analyzed to identify secret propagation flows in $\mathcal{S}'$. We log a taint sink point in $\mathcal{S}$ when encountering: 1) a tainted memory read/write or 2) a tainted branch condition. A statement is tainted if it is directly or indirectly dependent on the secret input, thus ensuring a comprehensive information flow modeling rather than merely data flow.

SideEval then compares two sets of taint sink statements $\mathcal{S}$ and $\mathcal{S}'$ to assess the security guarantee of the patch.[8] In particular, it checks if $\mathcal{S}' \subseteq \mathcal{S}$, indicating that $\mathcal{S}'$ contains less or equal sink statements than that of $\mathcal{S}$, and no secret propagation flow exists in $\mathcal{S}'$ that is not in $\mathcal{S}$. This is intuitive, as it is presumably unlikely that a patch will introduce extra new secret-dependent memory accesses or branches (which likely indicate new side channels). In case the above condition is violated, SideEval issues a warning. Like standard compiler warnings, the outputs of SideEval pinpoint problematic code, identify their causes (e.g., branches, memory accesses, or ciphertext dependencies), and prompt developers to re-examine the patch.[9]

Modeling — Instruction Cache Side Channels. Besides the holistic secret information flow check, we analyze concrete side-channel models. SideEval considers cache and ciphertext side channels for this study. The secret flow analysis discovered all secret-dependent memory accesses and control transfers on the sub-call graph $\mathbb{G}'$ composed of the patched function F' and its callees. Thus, we execute static symbolic execution on $\mathbb{G}'$ and verify secret-dependent branch conditions $C(i)$ using the following constraint [14]:

$$C(i) \neq C(i')$$

where i, i' are two secret inputs and $C(i), C'(i)$ are the symbolic constraints of branch conditions C on i, i'. Standard constraint solvers like Z3 [43] can verify this constraint. The constraint checks if a control transfer in F' (or its callees) depends on the secret. If the constraint solver deems the condition satisfied, attackers may use the ICache(instruction cache) side channel to infer the covered branch and secret i[4]. SideEval then reports a warning.

Modeling — Data Cache Side Channels. Similarly, SideEval considers the following constraint to check the security guarantee of a patch against data cache side channels:

$$M(i) >> L \neq M(i') >> L$$

[8] Two memory/branch instructions in $\mathcal{S}, \mathcal{S}'$ are deemed equivalent if they are from the same source code line or operating on the same source variables. This can be easily checked by compiling cryptographic code with debug symbols.

[9] Theoretically, it is not impossible that a patch introduces new secret propagation flows, yet the patch is still valid. However, to our observation, this is rare. In practice, enforcing this condition is effective to identify potential security issues in a patch, as we will show in Sec. 7.1.

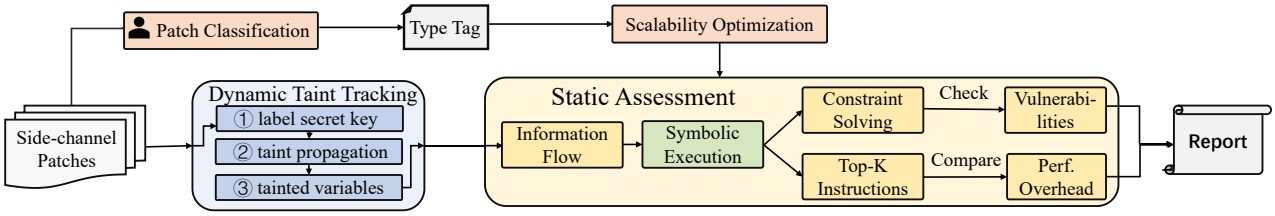


Figure 4: The workflow of SideEval.

where i, i' represent secret inputs and $M(i), M(i')$ represents the memory address. According to standard cache line and cache bank attacks, L can represent the cache line size in bits (6 in current x86 processors) or cache bank size in bits (2 as common size). The constraint [53] determines if two secret inputs can access separate cache units (cache line or cache bank). Attackers can use the DCache (data cache) side channel to deduce the accessed cache line/bank and secret input if the preceding condition is met. In that situation, SideEval will issue a warning.

Modeling — Ciphertext Side Channles. Further to the cache side channels checked above, SideEval considers the following constraint to check the security guarantee of a patch against ciphertext side channels:

$$W_1(i_1) = W_2(i_2) \wedge W_1(i_1') \neq W_2(i_2')$$

where $[i_1, i_2]$ and $[i_1', i_2']$ are two pairs of different secret inputs, and $W_1(i_1), W_2(i_2)$ denote the symbolic constraint of the written value in two consecutive memory writes toward one memory address. This constraint is from the recent work [16], and it aims to check if two secret memory access via the same piece of secret can result in ciphertext collisions. If so, attackers can monitor the ciphertext changes to infer the secret input. If the above condition is decided as "satisfied," SideEval reports a warning to the developer.

6.2 Performance Modeling and Analysis

Assessing the overhead of side channel patches is not easy, given that side-channel patches often influence multiple paths in a function, and certain paths are hard to trigger, e.g., only triggered when the RSA base is a small number (2). Therefore, conventional dynamic profiling is not desirable in assessing the performance of side-channel patches. On the other hand, while static instruction count (IC) [24, 25] is not impeded by the coverage issue, it only counts the number of assembly instructions without considering loops, branches, and jumps [33].

Performance Metric. SideEval conducts performance analysis with symbolic execution. Specifically, following the security analysis (Sec. 6.1), we employ symbolic execution for path discovery on the sub-call graph $\mathbb{G}'$ of the patched function F' and its callee functions. We then conduct a variant of IC on each path, which computes the number of CPU cycles necessary for a path. The computation is shown in Eq. 1, where $p = (Inst_1, Inst_2, \ldots, Inst_n)$ is a valid path of the patch code, $Inst_i$ is one executed instruction, $N_{cycles}(Inst_i)$ denotes the CPU cycles required by $Inst_i$, and $IC(p)$ denotes the total cycles required to execute the path p.

$$IC(p) = \sum_{i=1}^{n} N_{cycles}(Inst_i) \quad (1)$$

For every path discovered with symbolic execution, we gather the number of required CPU cycles. Therefore, given the sub-call graph formed by the patched function F' and its callees $\mathbb{G}'$ (where F' denotes the entry point of the sub-call graph), we assume that it has n paths, denoted as $(p_1, p_2, \ldots, p_n)$. At the performance analysis stage, we collect a set $S' = (IC(p_1), IC(p_2), ..., IC(p_n))$, where $IC(p_n)$ denotes the IC of the n_{th} path. The IC set S of the unpatched function can be computed similarly.

It is not easy to compare two sets of ICs, as the number of paths in S and S' may differ. We propose heuristics by first ruling out some paths whose associated path conditions have only a few solutions; such paths are likely error-handling paths that are rarely executed and their execution time can be ignored in performance evaluation. Given the general difficulty of statically deciding the most frequent path, we recast the problem into measuring the secret information flow on each path, and picking K paths with the top-K highest number of tainted instructions. We view this as a reasonable heuristic, as those paths with the highest frequency of secret propagation are likely to be the most sensitive paths, involving the majority of secret-dependent operations. As a reasonable setup, we set $K = 5$ to use the top-5 paths for comparison, and we discuss the influence of different K values in Section 8. To decide an instruction's CPU cycles, we directly refer to the Intel manual [2].

6.3 Scalability Optimization

Following the modeling in Sec. 6.1 and Sec. 6.2, SideEval conducts a comprehensive security and performance analysis on a patch. Despite its effectiveness, SideEval's analysis is expensive, given the complexity of in-the-wild cryptographic software. To alleviate the scalability problem, we classify patches into three types and apply different optimization strategies to trim off the sub-call graph in the context of a patch. Details could be found in the extended version [1], and we show relevant evaluation in Sec. 7.4.

7 Findings

With the collected in-the-wild patches in SideBench, we aim to answer four research questions (RQs): **RQ1**: How do in-the-wild side-channel patches perform regarding security effectiveness and introduced execution overhead? **RQ2**: What root causes led to the failures of these side-channel patches? **RQ3**: How is the patching capability of some emerging automated patching tools? Moreover, to illustrate the merit of SideEval, we answer **RQ4**: How does SideEval perform compared with prior relevant tools? and how do the optimizations implemented in SideEval affect the analysis results?

Table 4: Evaluation results. × denotes that the patch has cache/ciphertext side channels. Characteristics of insecure patches are denoted in the Category column. Time denotes how long in months the insecure patch existed in the codebase, and ◯ means that it has *not* been fixed by the time of this study. Superscript in Commit Id means that multiple patches have been committed in one commit history, e.g., 46a0ee8[1] and 46a0ee8[2] patched the functions `_fp_exptmod` and `wc_ecc_mulmod`, respectively.

Implementation	Commit Id	CVE	Type	Security		Performance	Category / Time (months)
				Cache Side Channels	Ciphertext Side Channels		
OpenSSL	f9b6c0b	2014-0076	③	×	×	1.3x	C / 48
OpenSSL	d6482a8	2016-0702	①		×	2.7x	B / ◯
OpenSSL	99540ec	2018-0735	①			1.4x	
OpenSSL	a9cfb8c	2018-0734	②			1.1x	
OpenSSL	c0caa94	*	②	×		1.3x	A / 11
OpenSSL	40e48e5	2018-5407	③		×	1.1x	B / ◯
OpenSSL	4a089bb	*	②	×		1.6x	A / 11
OpenSSL	972c87d	*	①		×	1.3x	B / ◯
WolfSSL	9d4fb79	*	③	×		1.1x	A / 20
WolfSSL	46a0ee8[1]	*	②	×		1.9x	A / 47
WolfSSL	f0db2c1	*	③		×	2.1x	B / 14
WolfSSL	0b2b218	2021-46744	③		×	1.3x	A / ◯
WolfSSL	0bcd38f[1]	*	①			1.1x	
WolfSSL	4f714b9	*	③		×	1.4x	B / 34
WolfSSL	0bcd38f[2]	*	①			1.4x	
WolfSSL	46a0ee8[2]	*	②	×		1.1x	A / ◯
MbedTLS	1297ef3	2020-10932	③		×	1.7x	B / ◯
MbedTLS	5b0589e	*	②			1.2x	

7.1 RQ1

Table 4 reports the findings over in-the-wild side-channel patches. We find that 25 out of 165 patches are insecure, and nearly all of them incur noticeably high overhead. Due to limited space, we report part of the data in Table 4 for discussion and list the full analysis for each patch in the extended version [1].

Insecure Patches. We characterize insecure patches into the following three categories:

A) Patches with Incomplete Fixes. We find that 11 out of 25 patches are incomplete. Incomplete fixes mean that the cache/ciphertext side channels still exist after the fix. *Nine* of 11 incomplete patches are repeatedly patched in later versions. For example, in `OpenSSL`, the developer incorrectly patched the `dsa_sign_setup` function (commit `c0caa94`), which is used to compute the DSA signature. Developers patched the function by mitigating the secret-dependent branches. However, the patch introduced another secret-dependent branch, thus leading to an incomplete fix. By checking the commit history, we find that it takes developers 11 months to realize this and eventually fix the unsafe patch. We list the time developers take to finally fix these incomplete patches in Table 4. As shown, all side channel leakages listed in Table IV survived a long enough time that they could potentially be exploited by attackers even after patching, and some led to new CVEs. We thus believe this finding reveals serious security flaws in the modern side channel patch development process, especially for cryptographic software. In contrast, SideEval can immediately identify the security issues in the patch and thus help the developer fix the patch promptly.

Furthermore, *two* out of 11 incomplete patches still exist in the codebase by the time of this study. For example, in `WolfSSL`, the developer attempted to patch the `_fp_exptmod_ct` function using a constant-time memory access pattern (commit `46a0ee8`). However, we find that the function is not "perfect" constant-time, in the sense that the function may leak the secret input through residue secret-dependent memory writes. We have reported the vulnerability to the WolfSSL developers. They prepared a new patch promptly, and committed to the codebase (after Ver. `5.6.3`). We believe the incomplete patches are due to the lack of systematic analysis tools to help the developers analyze patches. Manually analyzing side-channel patches is inherently challenging due to the complexity of side-channel vulnerabilities.

B) Patches Introducing New Kinds of Side Channels. We find that 13 of 25 patches introduce new kinds of side channels, i.e., when patching cache side channels, it may introduce new ciphertext side channel leakages. Consider Fig. 6 (details in Sec. 7.2), where the developer patched the `MOD_EXP_CTIME_COPY_FROM_PREBUF` function by adding constant-time operations. While this patch successfully mitigates the cache side channel, the patched function becomes insecure from the perspective of ciphertext side channels. The developers confirmed the finding and are preparing a new patch by the time of writing. In practice, the developers of `WolfSSL` have issued commits `0b2b218` and `0bcd38f` to fix ciphertext side-channel vulnerabilities introduced by patches `f0db2c1` and `4f714b9`. Overall, a number of patches can, in fact, introduce new side channels under different microarchitectural settings, which are subtle and hard to identify without the help of SideEval.

C) Incomplete Fixes & New Side Channels. We find that *one* patch fails to fix a known side channel leakage, and also introduces a new side channel. For example, in `OpenSSL`, the patch of `CVE-2014-0076` implemented the constant operation of `ECDSA` to resist the cache side channel attack. Nevertheless, the patch introduced the ciphertext side channel that enables an adversary to recover all secret key bits. Simultaneously, this patch is also incomplete, necessitating another patch to fix the `CVE-2018-5407` caused by the same leakage. Developers can utilize SideEval to assess the security of patches in terms of multiple side-channel modelings, and promptly eliminate side-channel vulnerabilities.

Performance Overhead. All analyzed patches have noticeable performance overhead, which is expected, as all side-channel patch paradigms introduce extra operations to obfuscate the attacker's observations to varying extents. For example, the constant-time

paradigm always adds redundant instructions to mitigate secret-dependent memory accesses, and the blinding paradigm strives to blind secret-dependent values using random masks. These extra operations lead to performance overheads. More than that, patched functions studied in this work are often involved in extensively executed loops (such as the iterative table lookup operations in RSA). These functions account for a significant portion of the program execution. The large number of loops allows these functions to incur substantial side channels that the attacker can observe. Similarly, ill-optimized patches in these functions are expected to cause significant observable performance degradation.

More interestingly, we find that the performance overheads vary significantly across different patches. In other words, we can identify inefficient fixes by cross-comparing the patched functions in different libraries. For example, in `OpenSSL`, the developer patched the modular exponentiation function in a rather verbose way, leading to a 2.7x performance overhead. In contrast, the same code patterns in `WolfSSL` are patched more efficiently, leading to only 1.1x performance overhead (more details in Sec. 7.2). In sum, we show that SideEval can identify "ill-optimized" patches (which exist in in-the-wild production cryptographic libraries like OpenSSL). It delineates the potential performance overheads of patches and thus helps developers optimize the patches more efficiently.

Answer to RQ1: A considerable number of side-channel patches are insecure or have overly significant performance penalties, even though they are issued by experienced cryptographic developers. This illustrates the commonness of side-channel patching errors, the significant challenge of writing side-channel patches, and the necessity of systematically measuring these patches.

7.2 RQ2

We examine each instance of unsafe patches and elaborate on the underlying causes for their security deficiencies. Simultaneously, we examine those ill-optimized patches to comprehend the factors contributing to the variations in performance overhead. We aim to provide concrete, actionable items for developers to avoid these pitfalls. Following, we summarize several suggestions for developers to write quality and secure side channel patches, where SideEval can be used as an automated tool to speed up this process.

Suggestion 1: *When patching the cache side-channel vulnerabilities, developers should comprehensively consider different patterns, including secret-dependent branches and memory reads/writes. Developers can leverage SideEval to check if all vulnerable patterns are considered and patched.*

Fig. 5 depicts a patch with dangling security flaws in WolfSSL. Function `_fp_exptmod_ct` executes the Montgomery ladder-based modular exponentiation. The variable `y` in this function denotes one bit of the private key, and it was found that the value of `y` (0 or 1) can be readily observed by attackers using cache side channels in unpatched versions [61]. The developers implemented constant-time memory reads in line 9 to mitigate the leakage disclosure of `y`. However, SideEval revealed that in line 6, the cache side channel remains as `R2` writes either `R0` or `R1` based on the value of `y`. Consequently, the attacker retains the ability to completely reconstruct the entire private key. The patch was implemented in September 2016, and the vulnerability has persisted in the WolfSSL repository for a duration of seven years. We reported the vulnerability to the developers, and they confirmed the finding. A new patch has been committed after being systematically analyzed using SideEval. For more details on our communication with the developers, please see Appendix A. We list general steps and guidelines for future research to effectively collaborate with cryptographic software developers when analyzing side-channel patches.

```
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.      ...
6.      fp_copy(&R[2], (fp_int*) ( ((wc_ptr_t)&R[0] & wc_off_on_addr[y]) +
7. ((wc_ptr_t)&R[1] &  wc_off_on_addr[y^1]) ) );
8.      // secret-dependent memory write
9.      fp_copy((fp_int*) ( ((wc_ptr_t)&R[0] & wc_off_on_addr[y^1]) +
10.                        ((wc_ptr_t)&R[1] & wc_off_on_addr[y]) ),
11.           &R[2]);
12.     err = fp_sqr(&R[2], &R[2]);
13.      ...  }
14.}
```

Figure 5: An unsafe patch for `_fp_exptmod_ct` in WolfSSL.

Suggestion 2: *When patching one microarchitectural side-channel vulnerability, developers should take account of other microarchitectural side channels, since the modified code may introduce another side-channel vulnerability, and even expand the leakage surface.*

Fig. 6 reports an unsafe patch in OpenSSL. As OpenSSL uses fixed window-based modular exponentiation, the function `MOD_EXP_CTIME_COPY_FROM_PREBUF` looks up the pre-computed table to obtain the corresponding exponentiation. The argument `idx` corresponds to several bits (usually 5 on 64-bit x86 platforms) of the private key. In the unpatched version (lines 1–9), variable `b` is written directly through querying the lookup table, which results in a cache side-channel vulnerability in line 7. Developers deployed a patch (lines 11–21) to eradicate the cache side channel. In this patch, rather than using a straight query, a circular process is implemented to read all the components of the lookup table for each `idx`. Then, variable `acc` is assigned conditionally, meaning that `acc` will be written multiple times, and each time when it is written, the value will change or remain unchanged. SideEval shows that the patch indeed causes a ciphertext side channel vulnerability. We reported this vulnerable patch and got it confirmed by developers.

Quantitative Analysis. To assess the leakage of patches, we analyze the above cache side channel and ciphertext side channel from a quantitative standpoint. In the unpatched program, each element in table `buf` is 4 bytes, and `idx` is 5 bits long. Thus, the basic unit of `buf` (`buf[0]`...`buf[31]`) is 128 bytes, which spans two cache lines. When a memory read `buf[idx]` happens, the attacker can observe that one among two cache lines is accessed, thus reducing the search space by half. Nevertheless, after patching, for each `idx`, the cycle (lines 16–18) will be executed 32 times, and only when `idx` equals `j`, `acc` is assigned with `table[j]`. At this point, ciphertext changes can be observed. Subsequently, attackers can infer the value in `idx`, using ciphertext collision attacks [35] and then recover the entire key. Overall, with quantitative assessment, we show that the patch failed to eliminate different microarchitectural side channels and expanded the secret leakage surface.

```
1. static int MOD_EXP_CTIME_COPY_FROM_PREBUF(BIGNUM *b, int top,
2.                  unsigned char *buf, int idx, int width) {
3.    ...
4.    for (i = 0, j = idx; i < top * sizeof b->d[0]; i++,
5.       j += width) {
6.    // write the pre-computed value to b directly
7.         ((unsigned char *)b->d)[i] = buf[j];
8.    ...}
9. }
10.
11. static int MOD_EXP_CTIME_COPY_FROM_PREBUF(BIGNUM *b, int top,
12.                  unsigned char *buf, int idx, int window) {
13.   ...
14.   for (int i = 0; i < top; i++, table += width) {
15.     BN_ULONG acc = 0; // temporary variable storing bignumber
16.     for (j = 0; j < width; j++) {    // conditional assign
17.       acc |= table[j] & ((BN_ULONG)0 -
18.                     (constant_time_eq_int(j,idx)&1));}
19.     b->d[i] = acc;  }  // write the pre-computed value to b
20.   ...
21. }
```

Figure 6: An unsafe patch for the table lookup in OpenSSL.

```
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],
12.                    R[set+0], a, modulus, mp);
13.    ecc_projective_add_point_safe(R[set+0],
14.           R[set+1], R[set+0], a, modulus, mp, &infinity) }
15.  ...
16. }
```

Figure 7: An incomplete patch for `ecc_mulmod` in WolfSSL.

Suggestion 3: *When patching ciphertext side channels, developers should place secret data into different addresses with the consideration of calling context. Since such a patching paradigm is hard (requiring inter-procedural analysis), we suggest developers use automated tools like SideEval to assist in patch development.*

Our observation shows that ciphertext side channels are hard to fix in practice. The code snippet of a patch for WolfSSL that was issued to mitigate ciphertext side channels is illustrated in Fig. 7. The function `ecc_mulmod` performs scalar multiplication using the Montgomery ladder algorithm. In contrast to the unpatched version, the patch uses data masking (line 5) and data moving (line 6) to fix ciphertext side channels. For data masking, it increments the variable `swap` during each iteration to ensure that the same ciphertext is never observed. For data moving, it duplicates the output of the conditional swap to distinct memory addresses. SideEval reports no defect in the data masking operations, as the value of `swap` changes for each iteration. However, SideEval flags the data moving operations as vulnerable because after conducting inter-procedural analysis, it is found that the callee functions invoked at line 11 and line 13 exclusively modify one of the two swapped variables. When the unchanged variable is returned to its original location, attackers can observe the same ciphertext and thus infer the relation of `swap` at the ith and $(i+1)th$ iteration. We reported the vulnerability to the developers, and they confirmed the finding. A new patch has been committed recently.

Suggestion 4: *Developers can cross-compare different patch implementations (in different libraries) to assess their patch efficiency.*

With SideEval, we compare patch implementations across different cryptographic libraries. Here, we present and compare the modular exponentiation operations in OpenSSL and WolfSSL: OpenSSL uses the fixed window-based modular exponentiation (Fig. 6), whereas WolfSSL uses the Montgomery ladder-based version (Fig. 5).

While both patches aim to achieve constant-time operations, their performance overheads differ. Fixed window-based modular exponentiation aims to mitigate the secret-dependent table lookup. This means that for each query, all entries in the table must be read and written redundantly to the intermediate variable (line 17 in Fig. 6), resulting in a 2.7x performance load. Nevertheless, in WolfSSL, the constant-time operation is implemented by selectively accessing either `R0` or `R1` (line 9 in Fig. 5), which merely introduces an extra & computation, causing a moderate overhead of 1.1x.

Answer to RQ2: Ensuring secure patching is challenging owing to the developers' erroneous assessment of various code patterns and the incorrect usage of patching schemes. The constant-time paradigms for cache side channels also give rise to ciphertext side channels, largely increasing the complexity of comprehensively patching vulnerabilities. Besides, different patching methods can result in significant variations in performance overhead. We provide several suggestions for developers on how to write high-quality and secure patches.

7.3 RQ3

Besides collecting and analyzing those in-the-wild side-channel patches, the community has been designing automated tools to generate side-channel patches. When expecting these tools to help developers automatically fix side-channel vulnerabilities, we were curious about the security guarantee and performance of patches generated by these tools. We conduct measurement study over two recent automated patch generation tools, Constantine [13] and Cipherfix [59]. Constantine is a compiler-based system that automatically hardens programs against cache side channels, where secret-dependent control and data flows are linearized. Cipherfix defends against ciphertext side channels by masking secret data with random values, ensuring ciphertext observations remain unpredictable. We used Constantine and Cipherfix to generate patches for ten vulnerable functions (see Table 8 and Table 9 in Appendix B for details). We then assess the security and performance of these patches using SideEval, with the results summarized in Table 5.

While Constantine successfully mitigated cache side channels, our findings reveal that it introduced ciphertext side channels in some cases when ensuring constant-time operations (similar to the patterns discussed in "Suggestion 2" of Sec. 7.2). This leads to 5 insecure patches, indicating that Constantine's patching rules lack systematic modeling for different side channel types. Cipherfix generated 3 insecure patches due to its incomplete information flow analysis. Moreover, Cipherfix's data masking approach imposes an average 4.5× performance penalty, while alternative strategies (e.g., data moving) cause minimal overheads (e.g., 1.1× overhead in the

WolfSSL patch of commit 0bcd38f). This suggests that data moving is a more efficient solution for fixing ciphertext side channels. In sum, we believe findings derived from this work provide guidance for improving automated patching tools, helping enhance both the security and performance of generated side-channel patches.

Table 5: Assessing automatically generated patches by Constantine and Cipherfix.

	Constantine (Cache)	**Cipherfix** (Ciphertext)
Insecurity Patches	5/10	3/10
Average Perf.	1.4x	4.5x

Moreover, given the recent success of large language models (LLMs) in comprehending complex code and generating patches for vulnerabilities [31, 46, 50, 62], we also evaluate LLMs' ability to patch side-channel vulnerabilities. We provide the configurations and prompts in the extended version [1]. Fig. 8 shows the patch snippet generated by GPT-4 when patching function `wc_ecc_mulmod` in WolfSSL 2.9.4. Compared with the unpatched version (see the extended version [1]), we find that the secret-dependent branch is correctly eliminated. Nevertheless, the LLM-created patch mistakenly uses a secret-dependent memory read, thus leading to a new cache side channel. Furthermore, we tentatively explored using LLMs to automatically generate patches for all 165 investigated cases. The results indicated that 63 of the generated patches incur compilation errors, preventing them from being applied directly. Additionally, 52 vulnerabilities were effectively resolved, while 50 cases remained vulnerable. Notably, among the 52 successful patches generated by GPT-4, 50 were for vulnerabilities disclosed and fixed before December 2023 (GPT-4's knowledge cutoff date), suggesting potential memorization rather than true patching. These findings highlight LLMs' limitations in vulnerability patching, reinforcing the need for SideEval. We leave it as future work to guide LLMs in patching zero-day vulnerabilities and expect a significant improvement when using more powerful LLMs in the future.

Answer to RQ3: Implementing automated side-channel patching tools still faces many challenges. We advocate the community to use SideEval with those tools to measure the quality of their generated patches.

7.4 RQ4

Efficiency of SideEval. To compare the detectability and efficiency with prior relevant tools[10], we run seven representative cache side channel detectors, CacheD [53], Abacus [10], CaType [27], CacheS [52], CacheAudit [17], MicroWalk [58], CacheQL [65], and one latest ciphertext side channel detector, CipherH [16], to analyze all 25 insecure patches found by SideEval. These 25 insecure patches contain 12 cache side channels and 14 ciphertext side channels (note that patch `f9b6c0b` in OpenSSL contains both cache and ciphertext side channels). Both Abacus and CacheD conduct symbolic execution towards an execution trace documented by Intel Pin, while Abacus is geared for analysis speed. CaType employs refinement types on a trace to track constant bit values. CacheAudit

[10] SideEval is the first tool for side-channel patch security verification, while other tools aim to locate side-channel vulnerabilities in the cryptographic libraries.

```
1.  int wc_ecc_mulmod(mp_int* k, ecc_point *G...) {
2.    ...
3.    for(;;){
4.       ...
5.       if (--bitcnt == 0) {
6.          if (digidx == -1) {
7.               break; }
8.           buf    = get_digit(k, digidx);
9.           bitcnt = (int) DIGIT_BIT;
10.          --digidx;
11.       }  // grab the next msb from the ltiplicand
12.       i = (int)(buf >> (DIGIT_BIT - 1)) & 1;
13.       ...
14.       // lines 7 and 10 are executed for both i == 0 and i == 1
15.      if (err == MP_OKAY)
16.        err = ecc_projective_add_point(M[0], M[1], M[2], modulus,
17.                                                          &mp);
18.      if (err == MP_OKAY)
19.        err = ecc_projective_dbl_point(M[1], M[2], modulus, &mp);
20.      // lines 13 and 16 are secret-dependent memory reads
21.      if (err == MP_OKAY)
22.        err = ecc_projective_add_point(M[0], M[1], M[i^1], modulus,
23.                                                          &mp);
24.      if (err == MP_OKAY)
25.        err = ecc_projective_dbl_point(M[i], M[i], modulus, &mp);
26.       ...}
27. }
```

Figure 8: The code snippet of a patch generated by `GPT-4` for function `wc_ecc_mulmod` in WolfSSL 2.9.4. Although the secret-dependent branch is correctly eliminated, the patch mistakenly uses a secret-dependent memory read.

and CacheS conduct static abstract interpretation over program call graphs. MicroWalk and CacheQL use mutual information to assess statistical correlations between secrets and memory accesses, therefore revealing cache side channels. CipherH uses (inter/intra-) symbolic execution and constraint solving to identify ciphertext side channels. Table 6 reports the results.

Table 6: Detectability and efficiency comparison with prior relevant tools. Time means the average time in minutes to analyze a patch.

	Cache	**Ciphertext**	**Performance**	**Time (min)**
CacheD	6	✗	✗	164.9
Abacus	8	✗	✗	89.7
CaType	8	✗	✗	158.4
CacheS	failed	✗	✗	failed
CacheAudit	failed	✗	✗	failed
MicroWalk	9	✗	✗	17.5
CacheQL	8	✗	✗	43.7
CipherH	✗	9	✗	205.4
SideBench	12	14	✓	2.3

Since all the related tools perform program-wide analysis rather than target a developer-specified patch point, they take much longer than SideEval for each patch. Meanwhile, none of them support the performance analysis of patches. Abacus, CacheD and CaType all launch trace-based symbolic execution over a tainted trace, which means they will inevitably miss some paths and, therefore, find fewer insecure patches. Moreover, production cryptosystems frequently involve hundreds of caller/callee functions on an execution trace, which accumulates symbolic formulas and increases formula sizes during analysis, making constraint solving more time-consuming. Nevertheless, SideEval conducts symbolic execution

from the entry of the patched function and chooses the inter-/intra-procedural analysis approach based on patch types, resulting in significant time savings for the analysis process.

CacheAudit and CacheS perform static abstract interpretation over a cryptographic library's call graph, which is inherently complex and less scalable. Moreover, when running CacheAudit and CacheS in these test patches, we see that they fail to support some instructions. With further exploration, we find new abstract operators must be defined in line with these instructions, which is challenging on our end (to define those operators properly and prove the soundness). We mark them as failed in Table 6.

MicroWalk and CacheQL locates cache side channels by feeding different secret inputs to execute programs and observing memory accesses. Although they have made a significant improvement in analytical time compared to prior methods (e.g., CacheD and Abacus), they still need to analyze a lot more irrelevant functions than SideEval. More importantly, some paths are hard to trigger by the inputs of MicroWalk, e.g., only triggered when the RSA base is a small number (2), thus causing fewer insecure reports. On the other hand, CipherH uses pattern-based inter-procedural analysis to improve scalability. However, insecure patches that are out of its considered patterns are overlooked, resulting in false negatives.

Effectiveness of Optimizations. We also study the effectiveness of optimizations implemented in SideEval (Sec. 6.3). Recall that we put patches into three categories and propose optimizations for each category; this is particularly designed to speed up the heavyweight static symbolic execution phase. We randomly selected two patch instances for each patch type to evaluate the effectiveness of the proposed optimizations. The results are reported in Table 7.

Table 7: Evaluation for the effectiveness of optimizations. "#callee" reports the number of callee functions that need to be analyzed for symbolic execution (security/performance). "time" denotes the duration for analysis. "failed" means that symbolic execution cannot finish in 30 minutes. ✗ denotes insecure patch. The last four rows denote statistics without optimizations.

	WolfSSL (0bcd38f)	OpenSSL (99540ec)	OpenSSL (a9cfb8c)	WolfSSL (46a0ee8)	OpenSSL (f9b6c0b)	MbedTLS (2ddec43)
type	①	①	②	②	③	③
#callee	0/0	0/0	0/4	0/2	1/2	2/2
security	✓	✓	✓	✗	✗	✓
performance	1.1x	1.4x	1.1x	1.9x	1.3x	1.2x
time	79s	473s	182s	167s	125s	211s
#callee	10/10	25/25	9/9	11/11	5/5	13/13
security	✓	failed	✓	✗	failed	✓
performance	1.1x	failed	1.2x	1.9x	failed	1.2x
time	389s	failed	423s	814s	failed	975s

Patch WolfSSL-0bcd38f (type ①) benefits from SideEval's optimization, which skips all its callee functions. Without this optimization, SideEval would need to analyze ten callee functions, increasing the analysis time significantly (from 79s to 389s, i.e., nearly five times longer). Similarly, for patch OpenSSL-99540ec, disabling optimizations prevents the analysis from completing within the 30-minute time budget due to the complexity of analyzing all 25 callee functions.

For patch OpenSSL-a9cfb8c (type ②), SideEval skips all its callee functions in the security analysis but analyzes only four modified callee functions in performance analysis. This optimization reduces the total analysis time from 423s to 182s while ensuring accuracy. In case of WolfSSL-46a0ee8, optimizations cut analysis time from 814s to 167s when identifying a specific vulnerability.

For patch OpenSSL-f9b6c0b (type ③), SideEval (with optimizations enabled) uncovers a security issue by analyzing only one newly introduced function in the security analysis. However, without optimizations, the analysis times out (exceeding 30 minutes) due to the overhead of handling five callee functions. Overall, we show that these optimizations enable the detection of a greater number of vulnerable patches in a reasonable time.

Answer to RQ4: Previous tools have substantially longer analysis time (or failed) and provide less thorough findings for patch assessment, since they and SideEval have distinct design considerations (whole-program analysis vs. patch-point analysis). Moreover, they are incapable of supporting performance analysis and can only identify a single kind of side channel. Also, studies show that the optimizations implemented in SideEval effectively improve efficiency. It helps uncover more vulnerabilities when analyzing side-channel patches in production cryptographic libraries under a given resource budget.

8 Discussion

Performance Measurement via Static Methods. Conventional dynamic performance profiling typically targets specific platforms to measure the program execution time. Nevertheless, the implementation of various optimization techniques by different processors, such as branch prediction and out-of-order execution, would have an impact on the actual runtime and analysis outcomes. Instead, we design SideEval to gather instructions and conducts performance analysis using symbolic execution, which is platform-independent and can generally adapt to changes in instructions and performance variations. Also, symbolic execution-based performance analysis can effectively trigger patch-modified code fragments. For example, commit `0bcd38f` fixes the function `_fp_exptmod_base_2` (inlined in `fp_exptmod`). The code modification is hard to trigger, making dynamic profiling challenging to assess the patch overhead. SideEval, through static path exploration, can cover the patched path and assess performance thoroughly. Also, SideEval delivers more detailed performance analysis results comparing to standard static IC, as explained in Sec. 6.2.

Extensibility. Our dataset SideBench can be extended with more data samples with different considerations. Analysis target-wise, we currently measure in-the-wild patches issued in cryptographic libraries. However, side-channel attacks can also be exploited in other domains, such as machine learning systems [65, 66]. Nevertheless, we notice that side-channel patches are not timely issued in these domains (e.g., to patch side channels in `libjpeg`). As disclosed in a relevant study [66], patching side channels in those domains often incur significant performance overhead. Looking ahead, we envision measure patches in those domains and help create efficient patches.

Modeling-wise, the current measurement focuses on modeling the general security information flow and two representative microarchitectural side channels (Sec. 6.1). Nevertheless, we can extend SideEval to measure other side channels, as long as proper side-channel modelings are provided, e.g., power side channels [42].

Implementation-wise, while our measurement is primarily conducted towards C/C++ implementations of cryptographic algorithms, we notice a growing trend of developing cryptographic software using high-level languages, such as Python, Java, and JavaScript [48]. Packages in `npm` are vulnerable to side-channel attacks and have been gradually patched in recent years [60]. We envision extending SideEval to measure side-channel patches issued in these languages. While the security and performance modeling can be reused, the underlying taint and symbolic analysis facilities need to be re-implemented, e.g., using popular engines like `Soot` [51].

Correctness and Comparison with Prior Patch Analyzers. Unlike functional patches that often add extra functional code to the original codebase, side-channel patches should *not* alter the original functionality of the codebase. Thus, it is feasible to check the correctness of side-channel patches by performing symbolic execution and asserting that no inputs can result in different outputs in the patched and original code. A representative work to detect the program's intended functionality is SPIDER [40]. We used SPIDER for the correctness check and found that all the analyzed side-channel patches are functionally correct. Nevertheless, we clarify that previous research, like SPIDER, has an orthogonal focus and cannot support analyzing side channel defects and performance, as we have proposed in this paper. In all, we believe "functionality correctness" is rather explicit and can be achieved by experienced developers. Nevertheless, security and performance are often more subtle and hard for developers to assert. Thus, we believe the current measurement focus is valuable and novel.

K in Performance Analysis. We set the default value of K to 5, as it consistently captures repaired paths across all analyzed patches. To validate this choice, we tested three additional configurations (K = 6, 7, and 8). The results, detailed in the extended paper [1], show negligible differences across different K values. Specifically, 148 out of 165 analyzed patches yield identical results (rounded to one decimal place) regardless of K, while the remaining 17 patches exhibit performance deviations within 14%. This justifies our experimental configuration (K = 5). Meanwhile, SideEval allows developers to adjust K for domain-specific needs if required.

Disclosure and Future CVEs. We responsibly disclosed all identified vulnerabilities to the affected library developers. This has already led to WolfSSL integrating two patches into their codebase, and we are actively collaborating with the OpenSSL and MbedTLS teams to address reported issues. As a measurement study, this research prioritized patch analysis and vulnerability assessment over exploit development. Consequently, our findings do not directly result in CVEs, as generating the requisite Proof-of-Concept (PoC) exploits was beyond the scope of this work and the current capability of SideEval. Nevertheless, the identified vulnerabilities have significant practical and security implications, regardless of CVE status or the challenges of automated exploitation, and these identified zero-day vulnerabilities can likely lead to new CVEs once PoC exploits are constructed.

9 Related Work

Software Side Channel Detection. While side channel attacks can severely tamper computer system security by stealthily leaking secret information from victim processes, some works aim to detect potential leakage sites in advance. CacheD [53] leverages symbolic execution and constraint solving to detect potential cache differences that cache side channel attacks may observe. CacheS [52] uses abstract interpretation to reason on program states with respect to abstract value statically. Abacus [10] tries to precisely quantify the leaked information in a single-trace attack with symbolic execution. CaSym [14] identifies side channels and provides sufficient information on where and how to mitigate them through preloading and pinning. CaType [27] analyzes cache side channels in crypto software using refinement type over x86 assembly code with cache layouts of potential vulnerable control-flow branches rather than cache states. CacheQL [65] quantifies side-channel leaks in production cryptographic and media software with mutual information and localizes the leak points in software with Shapley value. CiperH [16] mixes dynamic taint analysis and static symbolic execution to summarize symbolic constraints for secret-dependent store instructions and identify the existence of ciphertext side channels in cryptographic software.

Patch Analysis. Previous automated patch analysis work focused on analyzing security patches, i.e., patches fixing security vulnerabilities [47, 49, 54–56], and safe patches, i.e., patches that preserve application functionality [39, 40]. For example, GraphSPD detected security patches with Graph-based enriched code semantics [54]. Tian *et al.* [49] employed textual characteristics to detect security patches in Linux. Soto *et al.* [47] performed an extensive investigation on Java security patches and offered insights into automatic code repair within Java programs. However, they are unable to analyze the security of side-channel patches in cryptographic libraries, as they all lack modeling of side-channel vulnerabilities. SPIDER [40] and SPATCH [39] assess the safety of patches rather than the security, guaranteeing that downstream developers utilize these portable patches to ensure software supply chain security. As shown in Sec. 8, these works are orthogonal to ours, and the analysis results cannot provide side-channel security guarantees. Li *et al.* [34] conducted a comprehensive patch security analysis and proposed several general insights for enhancing the patching process. Nevertheless, their analysis process necessitates a substantial quantity of manual intervention and lacks automated tools to evaluate the security of newly committed patches.

10 Conclusion

We have presented the first systematic measurement on side-channel patches issued in production cryptographic libraries. We present a novel dataset, SideBench, that contains all side-channel patches issued in three mainstream cryptographic libraries, OpenSSL, WolfSSL, and MbedTLS. We also present SideEval, a scalable tool to facilitate automated patch measurement. Our study uncovers several insightful findings and subtle pitfalls, and provides guidelines for developers to write high-quality patches. Developers can integrate SideEval into their daily development workflow.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. The HKUST authors were supported in part by a NSFC/RGC JRS grant under the contract N_HKUST605/23. Zhibo Liu is additionally supported by an RGC PDFS grant (PDFS2324-6S08). Yinqian Zhang is in part supported by National Natural Science Foundation of China under grant No. 62361166633.

References

[1] [n. d.]. Research Artifact. https://github.com/Sen-Deng/SideEval.
[2] 2023. instruction cycles. https://www.agner.org/optimize/instruction_tables.pdf.
[3] 2024. angr. https://github.com/angr/angr.
[4] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New results on instruction cache attacks. In *International workshop on cryptographic hardware and embedded systems*. Springer, 110–124.
[5] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting secret keys via branch prediction. In *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*. Springer, 225–242.
[6] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 870–887.
[7] AMD. 2022. Technical Guidance For Mitigation Effects of Ciphertext Visibility under AMD SEV. https://www.amd.com/system/files/documents/221404394-a_security_wp_final.pdf.
[8] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 623–639.
[9] ARM. 2021. Arm Confidential Compute Architecture software stack. https://developer.arm.com/documentation/den0127/latest.
[10] Qinkun Bao, Zihao Wang, Xiaoting Li, James R Larus, and Dinghao Wu. 2021. Abacus: Precise side-channel analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 797–809.
[11] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).
[12] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision Timing Attacks Against AES. In *CHES*.
[13] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. 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*. 715–733.
[14] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. 2019. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 505–521.
[15] Jean-Sébastien Coron. 1999. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems: First InternationalWorkshop, CHES'99 Worcester, MA, USA, August 12–13, 1999 Proceedings 1*. Springer, 292–302.
[16] Sen Deng, Mengyuan Li, Yining Tang, Shuai Wang, Shoumeng Yan, and Yinqian Zhang. 2023. {CipherH}: Automated Detection of Ciphertext Side-channel Vulnerabilities in Cryptographic Implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6843–6860.
[17] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Sec*.
[18] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2015. Get your hands off my laptop: physical side-channel key-extraction attacks on PCs: Extended version. *Journal of Cryptographic Engineering* 5 (2015), 95–112.
[19] Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*. Springer, 444–461.
[20] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 368–379.
[21] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: Automating attacks on inclusive {Last-Level} caches. In *24th USENIX Security Symposium (USENIX Security 15)*. 897–912.
[22] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy*. IEEE, 490–505.
[23] Daniel Hedin and Andrei Sabelfeld. 2012. A perspective on information-flow control. In *Software safety and security*. IOS Press, 319–347.
[24] JL Hennessy and DA Patterson. 2012. Computer Organization and Design: The Hardware/Software Interface, Waltham.
[25] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.
[26] Intel. 2021. 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.
[27] Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. 2022. Cache Refinement Type for Side-Channel Detection of Cryptographic Software. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1583–1597.
[28] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. 2021. Supporting Intel SGX on multi-socket platforms. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html. (2021).
[29] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.
[30] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '96)*. Springer-Verlag.
[31] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch generation with language models: Feasibility and scaling behavior. In *Deep Learning for Code Workshop*.
[32] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-che Tsai, and Raluca Ada Popa. 2020. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 487–504. https://www.usenix.org/conference/usenixsecurity20/presentation/lee-dayeol
[33] Kevin M Lepak, Harold W Cain, and Mikko H Lipasti. 2003. Redeeming ipc as a performance metric for multithreaded programs. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 232–243.
[34] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2201–2215.
[35] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A systematic look at ciphertext side channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 337–351.
[36] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. {CIPHERLEAKS}: Breaking Constant-time Cryptography on {AMD}{SEV} via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*. 717–732.
[37] LLVM. 2020. DFSan. https://clang.llvm.org/docs/DataFlowSanitizer.html.
[38] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–37.
[39] Changhua Luo, Wei Meng, and Shuai Wang. 2024. Strengthening Supply Chain Security with Fine-grained Safe Patch Identification. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.
[40] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. Spider: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1562–1579.
[41] Hridoy Jyoti Mahanta and Ajoy Kumar Khan. 2020. A secured modular exponentiation for RSA and CRT-RSA with dual blinding to resist power analysis attacks. *International Journal of Information and Computer Security* 12, 2-3 (2020), 112–129.
[42] David McCann, Elisabeth Oswald, and Carolyn Whitnall. 2017. Towards Practical Tools for Side Channel Aware Software Engineering:'Grey Box'Modelling for Instruction Leakages. In *26th USENIX security symposium (USENIX security 17)*. 199–216.
[43] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proc. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.
[44] OpenSSL. 2022. Spectre and Meltdown attacks against OpenSSL. https://openssl-library.org/post/2022-05-13-spectre-meltdown/.
[45] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*.
[46] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair*. 69–75.
[47] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 512–515.
[48] sybrenstuvel. 2024. Pure Python RSA implementation. https://pypi.org/project/rsa.

[49] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 386–396.

[50] M Caner Tol and Berk Sunar. 2023. ZeroLeak: Using LLMs for Scalable and Cost Effective Side-Channel Patching. *arXiv preprint arXiv:2308.13062* (2023).

[51] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[52] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019. Identifying cache-based side channels through secret-augmented abstract interpretation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 657–674.

[53] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *USENIX Security*.

[54] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. GraphSPD: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2409–2426.

[55] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE, 595–600.

[56] Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2020. A machine learning approach to classify security patches into vulnerability types. In *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 1–9.

[57] Zhenghong Wang and Ruby B. Lee. 2006. Covert and Side Channels Due to Processor Architecture. In *ACSAC*.

[58] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MicroWalk: A Framework for Finding Side Channels in Binaries. In *ACSAC*.

[59] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. 2023. Cipherfix: Mitigating Ciphertext {Side-Channel} Attacks in Software. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6789–6806.

[60] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. 2022. Microwalk-CI: practical side-channel analysis for JavaScript applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2915–2929.

[61] WolfSSL. 2016. Cache side channel vulnrabilities in wolfssl. https://github.com/wolfSSL/wolfssl/commit/46a0ee8e690913a41a10f3c296cfab8345500218.

[62] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).

[63] Yuval Yarom and Katrina Falkner. 2014. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*. 719–732.

[64] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* 7 (2017), 99–112.

[65] Yuanyuan Yuan, Zhibo Liu, and Shuai Wang. 2023. CacheQL: Quantifying and Localizing Cache Side-Channel Vulnerabilities in Production Software. In *32nd USENIX Security Symposium (USENIX Security 23)*.

[66] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Automated side channel analysis of media software with manifold learning. In *31st USENIX Security Symposium (USENIX Security 22)*. 4419–4436.

[67] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 305–316.

[68] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 990–1003.

A Communication with Developers

Figure A outlines our communication process with developers. Specifically, when SideEval detects a side-channel security issue within a patch, the relevant details are communicated via email to the developers (①). Typically, developers acknowledge the report and propose a candidate fix (②). This revised patch is then re-evaluated using SideEval. If vulnerabilities are identified during re-evaluation, another report is dispatched to the developers. Conversely, if the patch passes validation, we confirm its security to the developers (③). Following this confirmation, developers usually proceed to commit the verified patch to the official code repository (④). In parallel with this process, the workflow continues by evaluating the subsequent patches from SideBench (⑤).

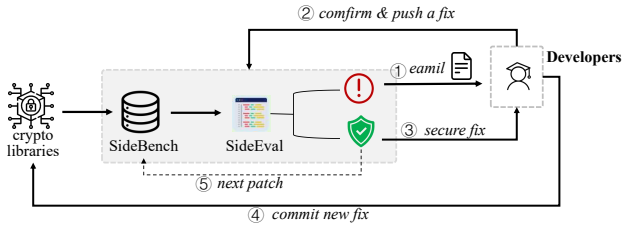


Figure 9: Communication with developers of cryptographic software.

B Assessment of generated patches

Table 8: The security and performance of patches generated by Constantine. ✕ denotes insecure patches.

Library	Function	Security	Performance
OpenSSL-5b820d7	`ec_wNAF_mul`	✕	1.2x
OpenSSL-56a98c3	`bn_cmp_words`	✓	1.1x
OpenSSL-5b820d7	`ossl_ecdsa_sign_sig`	✓	1.3x
OpenSSL-3c5a61d	`BN_num_bits_word`	✕	1.6x
WolfSSL-1f78297	`ecc_mulmod`	✕	1.8x
WolfSSL-e87ded8	`array_add`	✓	1.3x
WolfSSL-2a22179	`_fp_exptmod`	✕	1.7x
MbedTLS-96449ce	`ecp_gen_privkey_mx`	✓	1.4x
MbedTLS-22589f0	`mbedtls_mpi_cmp_mp`	✕	1.2x
MbedTLS-15f2b3e	`mbedtls_mpi_read_binary`	✓	1.3x

Table 9: security and performance of patches generated by Cipherfix. ✕ denotes insecure patches.

Library	Function	Security	Performance
OpenSSL-40e48e5	`ec_mul_consttime`	✓	3.5x
OpenSSL-972c87d	`BN_num_bits_word`	✕	4.7x
OpenSSL-22aa4a3	`DH_compute_key`	✓	5.2x
OpenSSL-8b44198	`BN_num_bits`	✓	4.3x
OpenSSL-f9b6c0b	`ec_GF2m_montgomery`	✕	5.1x
WolfSSL-f0db2c1	`ecc_mulmod`	✓	6.2x
WolfSSL-4f714b9	`_fp_exptmod`	✕	1.4x
WolfSSL-0b2b218	`ecc_mulmod`	✓	5.3x
MbedTLS-1297ef3	`mbedtls_mpi_exp_mod`	✓	4.6x
MbedTLS-ee6abce	`mbedtls_mpi_cmp_mpi_ct`	✓	5.1x