Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software

Jan Wichelmann*, Anna Pätschke*, Luca Wilke, and Thomas Eisenbarth

University of Lübeck, Lübeck, Germany

{j.wichelmann, a.paetschke, l.wilke, thomas.eisenbarth}@uni-luebeck.de

Abstract

Trusted execution environments (TEEs) provide an environment for running workloads in the cloud without having to trust cloud service providers, by offering additional hardware-assisted security guarantees. However, main memory encryption as a key mechanism to protect against system-level attackers trying to read the TEE's content and physical, off-chip attackers, is insufficient. The recent Cipherleaks attacks infer secret data from TEE-protected implementations by analyzing ciphertext patterns exhibited due to deterministic memory encryption. The underlying vulnerability, dubbed the ciphertext side-channel, is neither protected by state-of-the-art countermeasures like constant-time code nor by hardware fixes.

Thus, in this paper, we present a software-based, drop-in solution that can harden existing binaries such that they can be safely executed under TEEs vulnerable to ciphertext side-channels, without requiring recompilation. We combine taint tracking with both static and dynamic binary instrumentation to find sensitive memory locations, and mitigate the leakage by masking secret data before it gets written to memory. This way, although the memory encryption remains deterministic, we destroy any secret-dependent patterns in encrypted memory. We show that our proof-of-concept implementation protects various constant-time implementations against ciphertext side-channels with reasonable overhead.

1 Introduction

The current trend for data processing and provisioning of infrastructure heads towards cloud computing, with many co-located clients sharing the same physical hardware instead of working in isolated self-hosted environments. To protect different clients from each other, as well as the hypervisor from the clients, virtual machines (VMs) are used to provide isolation. However, especially when processing sensitive data, users may also want isolation from the hypervisor for data privacy or regulative reasons. This kind of isolation can be provided by trusted execution environments (TEEs), which model the hypervisor as an untrusted party. To achieve this kind of isolation, TEEs use a combination of additional access rights and cryptography to prevent the hypervisor, or more general, any privileged attacker, from reading the content of the TEE or interfering with its execution state.

Nevertheless, sharing the same hardware leads to traces in shared resources like caches which in turn provides an attack surface for timing or microarchitectural side-channels [6, 10, 28, 34, 40]. A widely used countermeasure against these side-channels is constant-time code that is data oblivious, i.e., does not access memory or decide for branch targets based on secrets [1, 52]. To support developers, there are various mostly automated constant-time analysis tools that observe different properties of software traces for finding microarchitectural or timing leakage that could lead to exploitable side-channels [1, 17, 49–52]. As these tools advance the constant-time properties of code, leakages get smaller and harder to find, though recent research has shown that even very small leakages are exploitable, especially when the strong attacker model of TEEs is considered [5, 36, 47].

The recent Cipherleaks paper [33] and its follow-up [31] introduced a new attack vector on code running in TEEs, dubbed the ciphertext side-channel. The core idea is that some TEEs use deterministic memory encryption, resulting in a one-to-one mapping between plaintexts and ciphertexts for a given memory block. As a result, the attacker can correlate changes in the ciphertext to the processed data. For example, the secret decision bit of a constant-time swap operation can be leaked by observing whether the ciphertext of the corresponding memory location changes, showing that state-of-the-art constant-time code is not secure under this attacker model. Thus, this attack vector demands for new analysis methods and countermeasures.

In this work, we introduce an analysis technique to mitigate ciphertext side-channel leakages in constant-time code. A naive approach hardening every memory write access would result in a very high performance overhead. Thus, our technique uses secret-tracking to pinpoint critical memory ac-

*These authors contributed equally to this work.

cesses, that are then safeguarded by randomizing observable write patterns such that the resulting binary does not leak information through the ciphertext side-channel. By combining static and dynamic approaches, we design a solution that covers all program components and works without recompilation.

1.1 Our Contribution

We present the CIPHERFIX framework, the first general-purpose drop-in mitigation for ciphertext side-channel-based leakages. This includes the following contributions:

- We propose an analysis technique based on dynamic taint analysis to find all secret-containing memory locations in constant-time binaries that are potentially vulnerable to the ciphertext side-channel.
- We employ dynamic binary analysis to locate stack variables and enable context-aware tracking of heap allocations, in order to support robust static instrumentation.
- We develop a mitigation technique, based on static binary instrumentation, that hardens the software binary across library boundaries without requiring recompilation and that provides three different security levels.
- We evaluate our proof-of-concept implementation of CIPHERFIX regarding performance and security on various primitives from four widely-used cryptographic libraries and discuss the effects of different mitigation approaches.

Our source code is available at https://github.com/UzL-ITS/Cipherfix.

Outline. After providing background in Section 2, we give an overview over the design of CIPHERFIX in Section 3. In Section 4, we present our dynamic analysis, which we use to build the static mitigation as described in Section 5. We evaluate the performance and security of our mitigation in Section 6. Finally, in Section 7, we discuss design decisions of CIPHERFIX and point out angles for future work.

2 Background

2.1 Secure Encrypted Virtualization

AMD Secure Encrypted Virtualization (SEV) is a trusted execution environment (TEE) that is designed as a drop-in solution to protect whole virtual machines. It encrypts the RAM content of the VM with an encryption key inaccessible to the hypervisor [26]. The latest iteration, SEV Secure Nested Paging (SEV-SNP) [2], prevents the hypervisor from remapping or modifying VM memory, thwarting attacks like [12, 20, 32, 37, 53]. For the memory encryption, SEV uses AES-128 in the XOR-Encrypt-XOR (XEX) [45] mode of operation, where a tweak value is XOR-ed before and after encryption. SEV derives the tweak values from the physical address of a 16-byte memory block and a random seed generated at boot time.

2.2 Ciphertext Side-Channel

The ciphertext side-channel was first introduced in [33] and later generalized to arbitrary memory regions and implementations in [31]. Both papers extract cryptographic keys from state-of-the-art constant-time cryptographic implementations running in SEV-SNP VMs. While the attack vector in [33] has been fixed on a firmware level [3], the attacks from [31] remain unaddressed. The core idea is exploiting the deterministic encryption at a fixed memory location, to leak information by precisely observing changes in the ciphertext and correlating them with the (known) executed code.

The authors of [31] introduce two attack variants: The *collision* and the *dictionary* attack. Both attacks exploit repeated write operations to the same memory address. The collision attack extracts information from observing the same ciphertext over multiple writes. One common example is the `cswap` pattern (Figure 1): A variable is always written, but depending on a secret decision bit the old or the new value is selected. While in the former case the deterministic ciphertext remains unchanged, in the latter case a new value is written, producing a different ciphertext. Thus, by observing the ciphertext of the memory location before and after the `cswap`, the attacker can immediately infer the secret decision bit. In the dictionary attack, the attacker does not only rely on collisions, but maps ciphertexts to (partially) known plaintexts. As the dictionary attack relies on repeating ciphertexts as well, mitigating the collision attack also mitigates the dictionary attack.

While the attacks above target values explicitly written to memory by the application, they can also be used to extract register values. For this, the authors of [31] exploit that the operating system running in the SEV-protected VM stores the user space register values upon context switches on the stack. This mechanism allows an attacker to extract secrets residing in registers by forcing context switches and observing the ciphertexts. However, the authors also describe how to fix this issue, by randomizing the stack layout.

2.3 Binary Instrumentation

Binary instrumentation allows modifying compiled programs without access to the source code. This is commonly used to insert new code that gathers information.

Dynamic binary instrumentation (DBI) gives the opportunity to include the architectural state by executing the analysis routines while the program is running. There are numerous DBI frameworks, e.g., Valgrind [39], Intel Pin [35], DynamoRIO [9] or DynInst [11]. The Intel Pin framework compiles and inserts analysis instructions at runtime through an x86 just-in-time (JIT) compiler. The code is processed

```
cswap(p, q, b):
  c = ~(b - 1);    // b = 0 -> c = 00...00
  t = c & (p ^ q);
  p ^= t;
  q ^= t;
```

(a) Constant-time swap of `p` and `q`, depending on bit `b`.

	Ciphertext of `p`	
`b`	before `cswap`	after `cswap`
0	`e4c80f2a`	`e4c80f2a`
1	`e4c80f2a`	`aa2f2a61`

(b) Ciphertext of `p`, before and after calling `cswap`.

Figure 1: `cswap` and resulting ciphertexts for the encrypted RAM accessible by the attacker. 1a shows the procedure of a constant-time swap. Depending on the value of a secret decision bit `b`, the values `p` and `q` are swapped (`b` = 1), or left as-is (`b` = 0). 1b shows the effect on the resulting ciphertext: If the ciphertext did not change, the attacker can infer that `b` = 0; if the ciphertext changed, the attacker learns that `b` = 1.

in units called *basic blocks*, which are defined as instruction sequences that have a single entry and exit point. Through a number of callbacks, a so-called *Pintool* specifies the analysis code to be inserted during JIT compilation. The original instructions and the analysis code are combined such that the instrumentation is transparent to the analyzed program.

Static binary instrumentation (SBI) results in a modified standalone binary that is obtained by the use of rewriting or redirecting techniques. The execution of an instrumented binary does not depend on an instrumentation framework, which means that the main overhead comes from the inserted analysis code [4]. However, static instrumentation struggles with analyzing indirect branches, shared libraries and dynamically generated code [29, 35]. There are different approaches for adding analysis code to the binary at specific instrumentation points and then redirecting the control flow, such that both analysis and original application code are executed in the right order. To avoid breaking references, the instrumented code can be put into a separate `.instrument` section. An instrumentation point then redirects execution to this section, either through software breakpoints via the `int3` [38] instruction and a custom signal handler, or through direct jumps via so-called *trampolines* [11, 23, 24]. It is possible to combine multiple approaches to minimize their shortcomings, e.g., by inserting 5-byte jumps where possible, and falling back to 2-byte jumps or `int3` when not enough space is available. An example of trampoline-based instrumentation is illustrated in Figure 8 in the appendix. Recent binary rewriting approaches further optimize the instrumentation through using available metadata for lifting [54] or symbolization of references [19].

2.4 Dynamic Taint Analysis

Dynamic taint analysis (DTA) tracks the flow of selected information through a program during code execution. The data to be tracked is marked as a *taint source*, and its propagation is defined through a *taint policy*. The policy also determines the *taint sinks* that can be reached by the data. All instructions that process secret data are considered for the taint propagation. Data flow tracking can be done in various granularities, whereby byte-level tracking is the most commonly used. For each memory location and register, there is shadow memory containing the taint label information, so the performance overhead is directly connected to the granularity. If too much data is marked as tainted, this is called *overtainting*; tainting too little data is referred to as *undertainting* [4, 27, 46].

A widely-used x86 taint analysis tool providing fast taint propagation based on Intel Pin is libdft [27]. In order to also support 64-bit binaries, libdft has been extended for VUzzer64 [44] and the AngoraFuzzer [14]. The data flow-based byte-level taint propagation in libdft64 is implemented through handwritten rules for every instruction class.

3 CIPHERFIX Design

We first give an overview of the generic design of our ciphertext side-channel countermeasure.

3.1 Attacker Model

We assume an attacker that tries to extract secret information from a TEE, that is protected with a deterministic block-based memory encryption with address-dependent tweaks. The attacker knows the exact binary which is executed by the victim, but cannot access secret data that is stored within the TEE. They have root access to the machine running the TEE and are able to read the entire encrypted memory, but cannot decrypt or modify it. Furthermore, the attacker can make use of a controlled channel that allows them to track and interrupt the code running inside the victim's TEE. This means that they can reconstruct the entire control flow of the targeted application and annotate it with snapshots of the corresponding ciphertexts in memory. One instance of such a scenario is a malicious hypervisor attacking a VM that is protected with AMD SEV-SNP. Finally, we assume that potential operating systems running alongside the targeted application inside the TEE do properly protect register values from ciphertext side-channels attacks, as discussed in Section 2.2.

3.2 Countermeasure Requirements

Our overall goal is to produce a hardened binary which does not contain leaking memory writes. The countermeasure should not only protect the targeted program itself, but all its dependencies as well, as leakage may span multiple libraries

(e.g., a crypto library calls `memcpy` in libc), and library developers are unlikely to widely adopt ciphertext side-channel countermeasures themselves. Finally, we target application developers who build code on top of third-party libraries and who do not have the necessary insight to manually fix leakages in those libraries. Thus, a drop-in solution with little manual interaction is desirable here.

There are two major approaches to this: One could either create a compiler extension that rewrites vulnerable memory accesses at compile time, or modify existing binaries through SBI. A pure compiler-based solution needs to recompile all dependencies, which is complex and requires manual intervention. A combination of DBI and SBI can work directly with the compiled binaries and, given sufficient coverage, accurately identify and harden vulnerable memory writes. For these reasons, CIPHERFIX aims for a binary instrumentation-based solution. The trade-off between binary vs. source-based approaches is further discussed in Section 7.1.

3.3 Protecting Memory Writes

In order to protect an existing binary from being attacked through a ciphertext side-channel, the content-based patterns of write accesses to memory have to be obscured. In [31], the authors propose various approaches for randomizing observed ciphertexts: First, by limiting reuse of memory locations through using a new address for each memory write; second, by interleaving data with random nonces; and third, by applying a random mask when writing data. The first approach uses the fact that different memory addresses get different tweak values in the memory encryption, but has a high overhead when applied outside of well-defined conditions. The second approach requires extensive changes to data structures, which has many pitfalls and needs to be done by the compiler. Due to lower overhead and higher practicability, we thus opt for the last approach, i.e., we add a random mask whenever an instruction writes secret data to main memory. We further discuss the different approaches in Section 7.3.

The masking of data takes place before memory writes and after memory reads. To store the masks belonging to a particular memory chunk (e.g., a C++ object), we allocate a *mask buffer* of the same size, so there is a one-to-one mapping of data bytes to mask bytes. When writing data, we generate and store a new mask, XOR it with the plaintext, and store the masked plaintext; when reading, we read the mask and then decode the masked plaintext. Note that we need to ensure that at no point non-encoded secret data is written to memory, so all decoding must be done in secure locations like registers.

3.4 Tracking Data Secrecy at Runtime

While masking all memory writes provides good protection, it comes with a high overhead. In fact, only a fraction of all memory writes relate to secret information: As we assume

(a) CIPHERFIX-BASE (b) CIPHERFIX-FAST

Figure 2: CIPHERFIX-BASE stores the secrecy information in a separate buffer, and uses it to decide whether a given mask byte should be applied or not. This allows to safely have non-zero mask bytes behind public data, as they are ignored if the corresponding secrecy bytes are zero. In contrast, CIPHERFIX-FAST stores this information directly in the mask buffer, i.e., a mask byte is zero iff the corresponding data is public.

that the implementation is constant-time, there is no secret-dependent control flow, so, for example, return addresses pushed onto the stack by function calls can be safely written in clear text. The same is true for the data structures used by the heap memory allocator to keep track of memory chunks. Finally, there may be a point where data is no longer considered secret, e.g., when sending a signature over the network. We thus aim to find and protect those instructions that actually deal with secret data. However, this is non-trivial, as there may be instructions that access both public and secret data, depending on the context (e.g., from `memcpy`).

Thus, we need a way to detect at runtime whether a given memory address should be considered secret, i.e., whether the data at that address is masked, and whether we should apply a new mask when writing to said address. We propose two approaches for storing this *secrecy* information (Figure 2): In the first approach, which we denote CIPHERFIX-BASE, we allocate another buffer of the same size as the mask buffer, called the *secrecy buffer*. In the second approach, CIPHERFIX-FAST, we encode this information directly into the mask buffer.

3.4.1 Storing secrecy information separately

In CIPHERFIX-BASE we allocate a buffer that holds the secrecy information for each memory location. If a byte is public, the corresponding secrecy byte is `0x00`; if a byte is secret, the secrecy byte is `0xff`. The secrecy buffer is initialized on allocation, and may be updated during the lifetime of the object. This construction allows us to read and update data without branching, as we can combine the secrecy value S with the mask M via a bitwise AND ($\otimes$), before applying it to the data via a bitwise XOR ($\oplus$): When reading, we compute $P = \hat{P} \oplus (M \otimes S)$, so we only decode the stored (potentially masked) plaintext $\hat{P}$ if the address is considered secret. For writing, we always generate and store a new mask, and then compute $\hat{P} = P \oplus (M \otimes S)$ for plaintext P. As we make no assumptions about the mask, this generally functions as a

one-time pad: The mask M is fully random and independent from the plaintext P, thus $\hat{P}$ is independent from P as well.

3.4.2 Storing secrecy as zero masks

By separating mask and secrecy information, CIPHERFIX-BASE can generate uniform masks, yielding a one-time pad encoding. However, this comes at a cost: First, we get high memory overhead by allocating the mask and secrecy buffers. Second, each read is replaced by three reads, namely to the data, mask and secrecy buffers. To reduce this overhead, we make an observation: If the data is public, ANDing the mask and the secrecy value yields zero; if the data is secret, we use the mask value directly. Thus, for CIPHERFIX-FAST, we merge the secrecy information and the mask into the mask buffer, by setting the mask to zero when the data is public, and to a random non-zero value otherwise.

For writes, we check whether the old mask is zero before generating a new one, saving a memory write in some cases; for reads, we directly XOR the mask value, saving a memory read compared to CIPHERFIX-BASE. Thus, in addition to the reduced memory overhead, we get a performance improvement due to fewer memory accesses. We discuss the security implications of this in Section 6.3.

3.4.3 Reducing risk of mask collision

While CIPHERFIX can be used with secret data of any size, the width of the masks influences the robustness against attackers that observe ciphertexts over longer periods of time. For example, for a $w = 8$ bit wide mask, a mask collision can be expected in as few as $\sqrt{2^w} = 16$ writes. To address this issue, we propose CIPHERFIX-ENHANCED, which, as an extension of CIPHERFIX-BASE, converts writes with a size w below a certain threshold to a bigger size w' that is considered safe: Instead of updating w bits, we generate a new mask of size w' bits and update w' data bits at once. This is possible due to architectures like x86 supporting multiple write sizes from 1 byte to 8 bytes (and even more with vector instructions). We thus read and decode the existing masked plaintext $\hat{P}'$ around the given address, merge it with the new plaintext P and then re-encode it. A write access protected with CIPHERFIX-ENHANCED is illustrated in Figure 3.

3.5 Toolchain

The CIPHERFIX framework is a drop-in solution that analyzes existing binaries with DTA to identify vulnerable code and then statically instruments the binaries to mitigate the detected leakages. CIPHERFIX consists of two distinct steps (Figure 4). In the analysis step, a taint analysis tool detects instructions and memory locations like stack frames and heap objects, that touch secret data. In parallel, a structure analysis tool extracts information about basic blocks and register/flag usage per

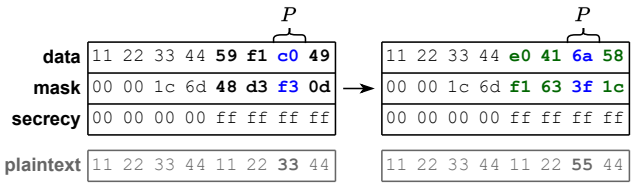

Figure 3: Extended write in CIPHERFIX-ENHANCED. Instead of updating only $w = 8$ data and mask bits at offset 6, CIPHERFIX-ENHANCED extends the write to $w' = 32$ bits, by also updating the mask of the surrounding three bytes, reducing the probability of a mask collision.

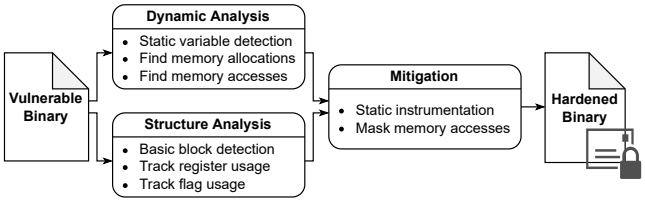

Figure 4: Structure of the CIPHERFIX framework. The vulnerable binary is dynamically analyzed and then hardened through static instrumentation.

instruction to aid the static mitigation. Finally, the mitigation step uses the analysis results to statically instrument the vulnerable binaries, inserting masking code for secret memory accesses and installing infrastructure for initializing newly allocated memory. In the following sections, we discuss the respective steps in more detail.

4 Leakage Localization and Preprocessing

In order to protect read/write operations, we first need to identify all vulnerable memory locations and the instructions accessing them. Our static mitigation relies on some additional structural information, i.e., the offsets of basic blocks and liveness of registers and flags. In the following, we describe our leakage localization technique and the other analysis steps.

4.1 Dynamic Secret Tracking

With the help of DBI and DTA, we can collect information that is only available at runtime. As constant-time code does not include secret-dependent control-flow, DTA covers all paths of the implementation. For the cases of non-constant control flow in public paths, we use multiple iterations of the program with different inputs. We further discuss this in Section 7.2. If an exact analysis is not possible, we stay on the safe side and avoid undertaining so that in combination with full path coverage we reliably identify all secret accesses.

Our proof-of-concept implementation is based on libdft64 data flow tracking. When combined with a Boolean taint, we

found that byte-level tainting of memory is fast enough to analyze complex cryptographic libraries while maintaining a high accuracy. While that leads to some overtainting (i.e., some memory locations get protected unnecessarily), we avoid undertainting. We also extended libdft64 by adding support for many SSE/AVX vector instructions, which are heavily used in optimized cryptographic code. All in all, we added 4,355 lines of code (LoC) to libdft64 for new instruction support and 2,269 LoC for our tracking logic.

4.1.1 Taint policy

We offer several venues for specifying taint sources, depending on the use case: First, if the main application itself can be easily recompiled (e.g., a custom network program linked against OpenSSL), the developer can call a special `classify` function, which takes a memory address and a size parameter. The taint analysis Pintool tracks this function and introduces taint for the corresponding memory when observing a call. In addition, we support fully automated assigning of taint sources without recompilation: Many cryptographic implementations read their private keys from the file system, so by intercepting the `open` and `read` system calls we can detect accesses to such files and taint the incoming data.

Our policy does not introduce taint sinks in the classical way; instead, those consist of all traced memory accesses and information that is needed for the static countermeasure. However, we offer a `declassify` function that explicitly marks data as no longer secret, i.e., all associated taint is deleted. In addition, functions that transmit data over insecure channels (e.g., network functions) remove taint as well. Thereby, we ensure that data that is meant to be publicly available does not get damaged by remaining secrecy features.

4.1.2 Tracking secret-related instructions

In order to protect memory accesses in our mitigation, we need to identify all instructions that read or write secret data at some point of the execution. The analysis distinguishes between three different cases: For instructions that only process public data there is no need to apply any ciphertext side-channel protection, whereas for instructions that only process private data the content written to memory always gets randomized. Finally, there are instructions that only occasionally access secrets and thus need to be able to distinguish between public and secret memory. As the latter may come with a certain performance overhead, the information about secrecy of accessed memory should be included in the taint analysis result used for the static mitigation.

4.2 Identifying Memory Locations

As the taint analysis itself tracks secrets only through "raw" memory addresses, it misses a lot of context: For example, there is no distinguishing between heap and stack memory, and which function a given accessed stack frame belongs to. However, for a static mitigation, we need certain information about each object in memory, like where it is allocated and which offsets need to be protected. There are various kinds of memory locations, i.e., static variables in the binary itself and dynamically allocated heap blocks and stack frames, so we need to distinguish between those cases.

4.2.1 Finding static variables

During the execution of cryptographic code, some instructions access data that lies within the memory region of the mapped binary, i.e., static initialized or uninitialized variables. Since we cannot access source-code level information about the program, we develop a method to locate these variables and determine their size, as we aim to only protect those that contain secret information. These fine-grained memory objects keep their secrecy status during the whole execution (i.e., if a variable contains secrets at some point, it is secret from the beginning until the end of a program run). For the static variable detection, we implemented a small Pintool with 338 LoC that collects traces of memory accesses in the data segments, matches these accesses to contiguous blocks in the binary's memory region and then produces an output file that can be parsed by the main taint tracking Pintool.

4.2.2 Heap allocations

Heap allocations are tracked through explicit (de)allocations, e.g., the `malloc`, `realloc`, `calloc` and `free` standard library functions. Similar to the static variables, the secrecy status of a heap object is kept for its entire lifetime. However, the heap layout may be different for each execution, so we cannot rely on fixed addresses to identify a heap object. Generating a flat list of heap allocations and retrieving the secrecy information using a counter variable is not useful either, as this restricts the hardened binary to a single control flow path. Instead, we use the call stacks of the heap allocations: Apart from rare cases where allocations are done in a loop, the call stack of each allocation is unique and thus suitable for identifying it both during analysis and at runtime. The call stack of an allocation is determined by keeping track of all calls and returns during analysis and emitting the current call stack whenever an allocation function is observed (Figure 5).

As for heap objects the application itself has full control over their layout and the stored data types may vary depending on context, we cannot safely make assumptions about relative offsets within a heap object. Thus, we opted for marking the entire object as secret whenever a part of it gets tainted. While this overapproximation may lead to a slightly higher overhead due to protecting more instructions than strictly necessary, it reduces complexity and makes the static mitigation more robust. We also found that in practice the impact is limited, as

Figure 5: Call tree and resulting call stacks for three heap allocations. The call stack is accompanied with secrecy information, i.e., whether a secret block was allocated. The offsets of call instructions that lead to at least one secret allocation are marked bold and red; the offsets of instructions that only lead to public allocations are marked green. This information is directly reused in the instrumentation (see Section 5.2.2).

generally the size of a heap object correlates with the amount of (private) data stored in it (e.g., big integer objects).

4.2.3 Tracking stack frames

The stack memory area is characterized by rather liberal (de)allocation and access strategies, which makes separating individual stack frames difficult and thus complicates tracking the exact offsets and lifetimes of secret variables. An easy solution would be marking the entire stack as secret and protecting all instructions that ever access stack memory, but this would introduce a lot of unnecessary overhead, since the stack is mostly used for temporarily storing registers and small local variables that often do not contain secret data. Instead, in order to avoid overtainting and the aforementioned performance penalty, we developed a generic stack frame tracking strategy that allows to keep track of secret data throughout the program execution by means of stack frame offsets. Contrary to the heap, the stack usually conforms to fixed patterns built by the compiler, so we can assume that relative offsets within a function's stack frame are valid over multiple executions.

Our proof-of-concept implementation does not rely on source code or function symbols, but works with any standard-conforming binary. The stack allocation tracking consists of identifying function calls, mapping a call target to an actual function for which a stack frame initialization of the static instrumentation is needed and determining its respective stack frame size, and building a list of secret offsets within that stack frame.

Most function calls are detected through `call`/`ret`-pairs; in addition, our analysis includes a heuristic for detecting tail calls, i.e., when a function is exited via a `jmp` instruction to another function. Calls to functions in shared libraries present another challenge, as the application invokes those through a call to the `.plt` section, which may in turn jump into the dynamic runtime linker to resolve the actual function call target. In order to find the function in the shared library and not its stub code in the caller's `.plt` section, we need to follow the resolution process in the dynamic linker until we reach the actual call target. This is done through a state machine that keeps track of the current linking state and generates a mapping of `.plt` offsets to the corresponding functions.

After detecting a function, we proceed with determining its stack frame size. This is achieved through several means: First, there may be explicit stack frame allocations through instructions like `push`/`pop` and `sub`/`add`, which directly modify the stack pointer. In addition, the x86-64 ABI permits functions to freely use a small chunk above the stack pointer (which usually marks the end of a stack frame), the so-called red zone. We handle this by updating the stack frame size whenever we observe an access outside a known stack frame.

4.3 Binary Structure Analysis

Contrary to DBI, where the executed code is recorded and instrumented at runtime, SBI must apply all changes in an offline manner, without being able to handle unexpected states. Our proof-of-concept SBI-based mitigation needs further information besides the DTA results, namely the precise bounds of all basic blocks and, for each instruction, the usage of registers and status flags. The latter is necessary since the masking operations need scratch registers to store intermediate results, and inadvertently clobber the status flags. While this information can be collected through static liveness analysis or heuristics [19, 54], we decided to employ dynamic analysis here as well, as we already have the necessary code coverage from the DTA. This approach marks only registers and flags that are indeed used, avoiding unnecessary saves/restores and thus reducing the runtime overhead. We created a specialized Pintool with 599 LoC that collects the aforementioned information and passes it to the SBI tool.

5 Static Mitigation

With the information from the dynamic analysis we can now statically instrument the affected binaries, hardening them against ciphertext side-channel attacks. We identify consecutive basic block chains (functions), which are then copied and instrumented at a new section in the binary. The original code locations are replaced by a number of jumps to their instrumented counterparts, following an optimized trampoline-approach described in Section 2.3. We then modify all vulnerable memory accesses to apply masks, such that each of these memory writes is randomized. The resulting hardened binaries are self-contained, i.e., they can be executed without an external instrumentation framework.

5.1 Masking Memory Accesses

After copying all affected basic blocks to a separate section, we can replace the vulnerable memory accesses by hardened

```
00: mov rcx, qword [rbp-0x20]        ; read encoded (?) data
04: mov rax, qword [rbp-0x3ffff020] ; read mask
0b: and rax, qword [rbp-0x2ffff020] ; AND secrecy value
12: xor rcx, rax                     ; decode (?) data
15: shr rcx, 8                       ; do actual computation
19: rdrand rax                       ; generate random mask
1d: jnc 19                           ; retry on failure
23: mov qword [rbp-0x3ffff020], rax ; store new mask
2a: and rax, qword [rbp-0x2ffff020] ; AND secrecy value
31: xor rcx, rax                     ; encode (?) data
34: mov qword [rbp-0x20], rcx        ; store encoded (?) data
```

Figure 6: Assembly code generated by CIPHERFIX-BASE for the instruction `shr qword [rbp-0x20], 8`, that accesses both public and secret memory. As an in-place shift, it has to first read and decode the left operand, compute the shift, and then encode and store the result. The instrumentation tool identified `rax` and `rcx` as scratch registers, which did not need to be preserved.

instruction sequences. As described in Section 3.4, we mitigate the ciphertext side-channel by adding a random mask to each memory write to a secret location. Some instructions have read and write accesses (e.g., arithmetic with a memory operand acting both as source and destination), so they may need decoding and encoding (Figure 6). String operations like `rep movsq` are replaced by an explicit loop that decodes each word of the source data and re-encodes it for the destination, as not the entire copied memory block may be secret. Our proof-of-concept implementation supports protection of common arithmetic and move instructions, and a number of vector instructions that occur in cryptographic code.

Each memory block is accompanied by a mask buffer and a secrecy buffer, which have a constant distance d_M resp. d_S to the memory block's address. Using a constant distance saves expensive look-ups for finding the appropriate buffers, reducing the total overhead of the mitigation. For our test setup, we found that $d_M = \texttt{0x3ffff000}$ and $d_S = \texttt{0x2ffff000}$ work well. These provide sufficient memory space while still fitting into the signed 32-bit memory displacement immediate which is supported by x86-64, and avoid penalties like aliasing when two addresses share too many low bits.

5.1.1 Updating the masks

Apart from initializing the mask and/or secrecy buffers during setup (see Section 5.2), we need to update the mask values before every write operation. To ensure that masks do not have repeating or easily exploitable patterns, we sample them from a pseudorandom number generator (PRNG). As we want to keep the overhead low, any such PRNG should have a small code footprint and require as few registers as possible, which rules out most classic software-based PRNGs. A natural choice on x86-64 is the `rdrand` instruction, which fills a single general purpose register with random bytes. The instruction offers cryptographically secure randomness. However, its security guarantees also lead to a noticeable slowdown when the instruction is used extensively.

To work around this, we devised two additional PRNGs for mask generation. The first one, named `AES`, makes use of the AES-NI `vaesenc` instruction to repeatedly apply the first round of AES to an initially random 16-byte state with a random 16-byte round key. The second PRNG is XorShift128+, a widely-used and fast full-period generator [48], for which we created a vectorized implementation. In both cases, the new mask is extracted from the state. For best performance, the `AES` PRNG needs two vector registers and the XorShift128+ PRNG needs three. We found that usually enough such registers are available, and, if not, the overhead for the additional save/restore is still smaller than calling `rdrand`. We discuss the properties of the different PRNGs in Section 6.3.1.

5.1.2 Scratch registers and flags

For some operations, we need additional scratch register space for storing intermediate results. Since we are restricted to working with an existing binary, we cannot exclude registers from being allocated by the compiler and thus have to look for registers which hold stale values, or save those values in a secure location. We use the results from the structure analysis in Section 4.3 to identify suitable registers. To save general purpose registers, we prefer using SSE vector registers via the `vmovq` and `vpinsrq` instructions, as those are fast and immune to ciphertext side-channels. In the rare case where no vector register is available, we store the scratch register's original value in memory. To avoid the expensive masking when writing a secret value to memory, we prioritize registers that the taint tracking did identify as not holding secret data.

Similar to the registers, our instrumentation may overwrite status flags through the encoding/decoding instructions. To save and restore single flags, we use the `set`*cc* instruction family, while for multiple flags we rely on the `lahf` instruction, which copies the entire flag state into the `ah` register.

5.2 Managing Mask and Secrecy Buffers

The instrumented instructions assume that there is a mask buffer and a secrecy buffer with a constant distance to the accessed memory address. Thus, for each memory block that is accessed by such an instruction, we need to allocate a mask buffer at the corresponding address and initialize it with random data, if it contains secret data. This comes with a few challenges: First, there are several ways of allocating memory, namely the stack, the heap and static fixed-size arrays in the binary itself. Then, not all memory blocks in these regions are considered secret, so their masks and secrecy values need to be initialized context-aware. In the following, we discuss strategies for handling the various memory regions.

5.2.1 Stack

The stack is allocated by the operating system at application start and is used for storing return addresses, register values and small local variables. The taint analysis produces stack frame information for each function, which contains the size of the stack frame and the relative offsets where secret data is stored. Accordingly, we insert a small code gadget at the beginning of each function, that prepares its stack frame by generating a random mask or setting the secrecy value for the respective offsets. The mask and secrecy buffers for the stack are allocated on startup; the constant buffer distances work well for the stack, as it usually resides within a well-known memory range and does not grow beyond a few megabytes.

5.2.2 Heap

For most Linux applications, the heap is a contiguous memory region that is managed by the standard library's allocator. The heap starts at a random base address, and is resized via the `brk` system call. The user then typically allocates memory by calling `malloc` or `realloc`, which ensure that enough heap memory is available and return an appropriate memory range.

To guarantee that there are mask and secrecy buffers backing the entire heap region, we instrument the `brk` system call and (de)allocate corresponding memory each time the heap grows or shrinks. The buffers are initially set to zero. We also replace the `malloc` and `realloc` calls by custom code, which ensures that the corresponding mask and secrecy buffers are correctly initialized depending on whether the allocated memory should contain secret data or not. To identify the particular heap allocation, we resort to tracking its call stack, as explained in Section 4.2.2. We achieve this through an *allocation tracker*, which is an integer residing at a fixed memory address, and which is updated on each `call` instruction that is part of a call stack that leads to a heap allocation. Before each `call`, we left-shift the tracker variable, and add 1 if the call is part of a call stack that leads to allocation of a secret heap memory object. With our allocation tracker, we can reliably handle heap allocations even if we encounter non-constant control flow or when a function is reused in a different context. An example is illustrated in Figure 7.

Contrary to `malloc`, the `realloc` function allows resizing or reallocating an existing heap memory object, while keeping its contents. As the new object may have a different secrecy setting than the old one, we have to ensure that the data is correctly decoded, copied and encoded. However, `realloc` itself is not aware of the masks and secrecy information, so to avoid losing information, our `realloc` handler copies the old data, mask and secrecy buffers to a separate memory location, runs `realloc`, and then restores the contents at the new location with the appropriate encoding.

If the instrumented program allocates lots of memory, the constant distance to the mask and secrecy buffers may be insufficient, as the heap could at some point overlap with its mask buffers. In this case, one could replace the affected `malloc` calls by a custom allocator, that is injected into the instrumentation and operates outside the usual heap area. Note that this still limits the maximum memory object size to the distance between a memory address and its buffers, i.e., at most two gigabytes, if the instrumentation should do without another scratch register for computing larger offsets.

Figure 7: Allocation tracking for the example from Figure 5. Each time a `call` instruction is executed, the allocation tracker is shifted to the left, and 1 is added when this particular `call` is part of a call tree leading to an allocation of a secret heap object. On return, the tracker is shifted back to the right. The `malloc`/`realloc` handler code then checks whether the allocation tracker has the value $2^n - 1$, i.e., whether it is all ones starting with the least significant bit. In this case, the new heap object is considered secret; else, it is public.

5.2.3 Static arrays

Finally, the binary may have a number of static global variables, which reside in its data sections. We embed the information about static memory objects containing secret data in the instrumented binary. On startup, an initialization routine walks through this list and allocates and initializes the respective mask and secrecy buffers.

5.3 Implementation

We created a proof-of-concept implementation of our mitigation in C#, which takes the dynamic analysis results and the target program and produces statically instrumented binaries. The instrumentation tool has 7,346 LoC, which includes a specifically developed library for patching ELF64 files.

5.3.1 Instruction instrumentation

The instrumentation tool loads and parses the outputs from the taint tracking and the structure analysis tools, and decodes the target ELF files. Then, for each individual binary, the instrumentation is applied: First, we look for contiguous basic block chains and identify appropriate code locations for inserting jumps to instrumentation code. Next, we replace each memory accessing instruction marked by the DTA by a masked version. After handling all basic blocks, we obtain a list of

unmodified and instrumented instructions, grouped by their respective basic blocks. In a final step, we re-assemble those instructions and write them into a newly allocated ELF section, while patching the basic blocks in the old `.text` section to jump to the instrumentation code.

5.3.2 Initialization

After the instruction-level instrumentation is done, we need to install infrastructure for handling the `int3` signals and some initialization code that allocates mask and secrecy buffers. For this, we created an *instrumentation header*, which consists of 966 lines of assembly code interleaved with some static constants which are later replaced by the instrumentation tool. The instrumentation header hooks into the *constructor* of each binary, which is executed by the dynamic linker when a binary is loaded into memory. This way, we ensure that our initialization runs before all other application code. The initializer of the main program sets up the signal handler, and determines the stack size and base address. Then, it allocates mask and secrecy buffers for the stack. The initializers of the main program and of all dynamic libraries iterate through the list of secret static variables deposited by the instrumentation tool, and allocate and initialize mask and secrecy buffers.

6 Evaluation

We now evaluate the performance and security of the different CIPHERFIX variants. We analyze whether there is remaining leakage with regard to collision attacks, and discuss trade-offs between security and performance.

6.1 Experimental Setup

We evaluate our proof-of-concept implementation of CIPHERFIX against a number of typical algorithms which are used in widespread protocols like TLS or SSH. To observe variations caused by different implementations of the same primitive, we spread our analysis over several common libraries, that are `OpenSSL` 3.0.2, `WolfSSL` 5.3.0, `mbedTLS` 3.3.0 and `libsodium` 1.0.18. As primitives which were shown to be vulnerable to ciphertext side-channel attacks [31], we picked EdDSA (Ed25519) and ECDSA (secp256r1), and verified that these are still vulnerable in the given implementations. [33] demonstrated an attack against the RSA signature scheme, which we included as well. We also added ECDH (X25519) as a primitive that is widely used in cryptographic protocols and likely to be vulnerable as well. As additional benchmarks, we included the symmetric primitives AES-GCM and ChaCha20-Poly1305, the hash function SHA-512, and finally the Base64 decoding function as a non-cryptographic algorithm, that is nevertheless often present in cryptographic applications.

The analysis, instrumentation and all measurements were performed on an AMD EPYC 7763 CPU with Zen3 microarchitecture, which supports SEV-SNP. All libraries were compiled with GCC 9.4.0 on Ubuntu 20.04.4 LTS. `MbedTLS` was linked statically, while the other libraries were linked as shared libraries.

6.2 Performance

To get the information necessary for the mitigation, we ran the dynamic analysis as described in Section 4. We found that we achieve sufficient coverage by executing each target 10 times with random inputs in a loop, except for WolfSSL RSA, which required 20 due to high control flow variation introduced by blinding. In all cases, the time required for dynamic analysis was less than 5 minutes, with around 80% of the time taken by the register tracking in the structure analysis, and most of the remaining time by the taint tracking. The most expensive target, `mbedTLS` ECDH, required tracking 170,532,009 executed instructions (5,167 unique). While the register tracking could be scrapped in favor of a faster but potentially less precise static liveness analysis (as done by several binary rewriting tools), note that these steps are executed offline and only need to be done once to protect a binary, so we deem an analysis time of a few minutes acceptable.

6.2.1 Runtime overhead

To measure the runtime overhead of the different CIPHERFIX variants, we executed each target with 1,000 random inputs, averaged the measured execution times, and computed the relative overhead compared to the original implementation. An overview of the resulting overall slowdowns of the different CIPHERFIX variants is given in Table 2. As expected, CIPHERFIX-FAST has the lowest overhead, CIPHERFIX-ENHANCED has the highest, and CIPHERFIX-BASE lies in between. The slowdown of CIPHERFIX-BASE compared to CIPHERFIX-FAST is caused by the additional read for each protected memory access; in most cases, CIPHERFIX-ENHANCED performs quite similar to CIPHERFIX-BASE, except for the symmetric primitives and utility functions which have a vastly higher number of 1-byte writes.

Moreover, generating masks with `rdrand` introduces a much higher overhead than with one of the other PRNGs. This is caused by the continuous reseeding of the underlying shared hardware PRNG, in combination with `rdrand` not being designed for sampling random numbers at a high frequency. The smallest overhead is achieved with the `AES` PRNG, as it consists of a single `vaesenc` instruction and only needs two vector registers. A detailed overview over all runtime overhead measurements is given in Table 1.

6.2.2 Code properties contributing to overhead

We identified several major factors that determine the overhead when hardening a particular implementation with CIPHERFIX. First of all, code that heavily relies on memory

Table 1: Runtime overhead of instrumented binaries. For each CIPHERFIX variant and PRNG, we measured the execution time in milliseconds (ms) of 1,000 executions of each primitive and the corresponding overhead factor to the original implementation. The target `AES` refers to AES-GCM, the target `CC20` to ChaCha20-Poly1305. The last row shows the geometric mean of the respective overheads for each CIPHERFIX variant.

	Target		orig	CF-FAST			CF-BASE			CF-ENHANCED		
				AES	XS+	rdrand	AES	XS+	rdrand	AES	XS+	rdrand
libsodium	EdDSA	time	29	159	166	1,159	189	248	1,133	214	245	1,134
		factor	-	5.5x	5.7x	40.0x	6.5x	8.6x	39.1x	7.4x	8.4x	39.1x
	SHA512	time	9	14	20	196	21	22	194	22	25	194
		factor	-	1.6x	2.2x	21.8x	2.3x	2.4x	21.6x	2.4x	2.8x	21.6x
mbedTLS	AES	time	104	297	377	2,849	364	371	2,576	1,204	1,213	2,683
		factor	-	2.9x	3.6x	27.4x	3.5x	3.6x	24.8x	11.6x	11.7x	25.8x
	Base64	time	10	12	13	58	16	16	45	28	30	46
		factor	-	1.2x	1.3x	5.8x	1.6x	1.6x	4.5x	2.8x	3.0x	4.6x
	CC20	time	144	324	332	2,945	542	570	2,952	1,785	1,721	3,059
		factor	-	2.3x	2.3x	20.5x	3.8x	4.0x	20.5x	12.4x	12.0x	21.2x
	ECDH	time	1,855	3,674	3,778	8,559	9,425	9,440	14,419	9,926	10,208	14,827
		factor	-	2.0x	2.0x	4.6x	5.1x	5.1x	7.8x	5.4x	5.5x	8.0x
	ECDSA	time	472	3,367	3,558	8,920	3,912	3,929	8,297	4,265	4,301	8,374
		factor	-	7.1x	7.5x	18.9x	8.3x	8.3x	17.6x	9.0x	9.1x	17.7x
	RSA	time	896	3,276	3,777	28,886	5,436	5,339	27,148	5,527	5,663	27,208
		factor	-	3.7x	4.2x	32.2x	6.1x	6.0x	30.3x	6.2x	6.3x	30.4x
OpenSSL	ECDH	time	172	541	550	2,408	664	657	2,323	708	807	2,369
		factor	-	3.1x	3.2x	14.0x	3.9x	3.8x	13.5x	4.1x	4.7x	13.8x
	ECDSA	time	516	939	1,121	7,181	1,795	1,855	9,980	2,051	1,973	10,072
		factor	-	1.8x	2.2x	13.9x	3.5x	3.6x	19.3x	4.0x	3.8x	19.2x
WolfSSL	AES	time	147	268	269	793	400	403	880	397	402	879
		factor	-	1.8x	1.8x	5.4x	2.7x	2.7x	6.0x	2.7x	2.7x	6.0x
	CC20	time	167	428	432	2,787	596	630	2,802	1,157	1,242	2,874
		factor	-	2.6x	2.6x	16.7x	3.6x	3.8x	16.8x	6.9x	7.4x	17.2x
	ECDH	time	146	258	437	4,217	544	565	4,070	541	558	4,070
		factor	-	1.8x	3.0x	28.9x	3.7x	3.9x	27.9x	3.7x	3.8x	27.9x
	ECDSA	time	1,092	1,704	1,954	15,765	3,945	3,834	18,631	3,883	3,897	19,654
		factor	-	1.6x	1.8x	14.4x	3.6x	3.5x	17.1x	3.6x	3.6x	17.1x
	EdDSA	time	60	124	156	1,897	279	265	1,759	280	290	1,761
		factor	-	2.1x	2.6x	31.6x	4.7x	4.4x	29.3x	4.7x	4.8x	29.4x
	RSA	time	133	248	334	2,901	588	605	2,863	602	651	2,870
		factor	-	1.9x	2.5x	21.8x	4.4x	4.5x	21.5x	4.5x	4.9x	21.6x
	average factor		-	**2.4x**	**2.7x**	**16.8x**	**3.9x**	**4.0x**	**17.3x**	**5.1x**	**5.3x**	**17.5x**

Table 2: Performance measurements for the different CIPHERFIX variants and PRNGs. Each entry shows the geometric mean of the runtime overhead over all targets compared to the original, uninstrumented binary.

	AES	XS+	rdrand
FAST	2.4x	2.7x	16.8x
BASE	3.9x	4.0x	17.3x
ENHANCED	5.1x	5.3x	17.5x

accesses for dealing with secret information is clearly more susceptible to overhead introduced by instrumentation than code that performs most computations in registers. This becomes apparent when comparing the RSA implementations of `mbedTLS` and `WolfSSL`: Though for `WolfSSL` a higher percentage of the memory accesses is instrumented (78% writes vs. 65%), `mbedTLS` has an order of magnitude more memory operations than `WolfSSL` and thus gets a higher overhead. Similarly, some instructions are more expensive than others in terms of ciphertext side-channel hardening: For example, arithmetic directly applied to memory operands requires a full decoding and re-encoding cycle (cf. Figure 6), which is slow due to direct data dependencies between the steps. We observed this for `mbedTLS` ECDSA, which gets a much higher overhead (7.1x vs. 1.6x) than the comparable implementation in `WolfSSL`, mostly due to expensive `adds` in a hot code path.

Finally, the overhead is influenced by the general structure of the instrumented code, and the optimization capabilities of the binary rewriting framework. A framework operating at basic block level could perform better than our proof-of-concept implementation, which instruments each instruction in isolation to ease leakage analysis and debugging. For example, scratch registers may not need to be restored between usages, and instructions could be reordered to avoid saving status flags. This is particularly relevant as the compiler tends to interleave arithmetic instructions that have direct status flag dependencies with memory accesses (e.g., `add-mov-adc`).

A detailed overview over the observed memory accesses is given in Table 3 in Appendix B.

6.3 Security

In the following, we illustrate reasons for remaining collisions after applying the different variations of CIPHERFIX and evaluate its practical security.

6.3.1 Leakage sources

As we assume full path coverage of the implementation (see Section 7.2) and our taint tracking does not undertaint, all vulnerable instructions are identified and protected. Thus, the only remaining source of leakage are **collisions of the masks or the masked plaintexts**: With CIPHERFIX-BASE, the secrecy information is stored in a separate buffer. If the mask M is fully random and independent from the plaintext P, the masked plaintext $\hat{P}$ becomes independent from P as well. However, the attacker can access both ciphertexts $C_{\hat{P}} = \mathrm{Enc}_{\mathrm{pt}}(\hat{P})$ and $C_M = \mathrm{Enc}_{\mathrm{mask}}(M)$, so they are able to detect whether $\hat{P}$ or M appear repeatedly. If the data memory block is rarely changed and the number of protected bits is sufficiently low, a mask collision is possible and may leak information about the plaintext. A similar issue can occur with CIPHERFIX-FAST, which stores the secrecy information directly in the mask buffer by setting the mask to zero for public values. We can assume that the attacker knows the ciphertext $C_0 = \mathrm{Enc}_{\mathrm{pt}}(0)$ of an unmasked zeroed data block, as memory usually is zero initialized. If they observe C_0 again after a write of P with mask $M \neq 0$, they can use that $C_0 = \mathrm{Enc}_{\mathrm{pt}}(P \oplus M)$ and thus $P = M$ to infer that $P \neq 0$. These leakages through masks or masked plaintexts are mostly relevant for 1-byte writes to variables in memory blocks with little other activity. With CIPHERFIX-ENHANCED, we enforce a minimum width of masked data, which further reduces the probability of mask collisions and other non-unique writes at the cost of a slightly higher overhead.

Another factor is the **quality of the PRNG used for mask generation**, for which we identified two primary criteria. First, the pseudorandomness should not correlate with the plaintexts: For example, simply incrementing the masks may lead to many collisions of the masked plaintexts in algorithms that use linear arithmetic. Second, deterministic PRNGs should have a sufficient cycle length, to keep an attacker from reliably triggering the same mask at the same address during the application's runtime. `Rdrand` offers the fastest available solution for cryptographically secure pseudorandomness. However, given the subsequent memory encryption, the used PRNG does not necessarily need to be cryptographic, as long as it satisfies the above criteria and thus does not tend to generate repeating masks or masked plaintexts. XorShift128+ has a cycle length of $2^{128} - 1$ and passes all *BigCrush* tests of the *TestU01* suite [30], though it has some weaknesses [22]. Our custom one-round `AES` PRNG passes all *BigCrush* tests and seems to perform well in practice, but does not have a guaranteed cycle length. We leave this analysis to future work.

6.3.2 Observed collisions

To analyze potentially remaining ciphertext collisions, we extended the taint tracking to export a full trace of all memory writes alongside corresponding secrecy information. We then created a Pintool that generates a trace of all memory writes for an instrumented binary. As each original memory access may be replaced by multiple memory accesses during instrumentation, we inserted special marker instructions that denote the beginning and end of a particular instrumented memory access sequence. With this information, we align the traces using a custom evaluation tool, and proceed with checking

whether there are repeated writes of the same secret value to the same address. As the dictionary attack builds upon the collision attack, finding no collisions implies security against all known ciphertext side-channel attack primitives. We found that using the same amount of test cases for our evaluation as for the initial taint tracking was sufficient, as due to the size and complexity of the evaluated targets systematic issues already appear during the first few executions.

We were able to confirm the suspected remaining leakages with our evaluation. For example, there are several thousand collisions for CIPHERFIX-BASE and CIPHERFIX-FAST with the `mbedTLS` AES-GCM target, which encrypts 16 KiB of plaintext using AES-NI and has 812,120 1-byte writes, which is 66% of its total writes. The observed collisions both included repeating masks and cases where applying a new mask to a new plaintext led to the same result. All colliding 1-byte writes were related to sequential writing into an array, e.g., when data is copied or buffers are cleared between different processing steps. The corresponding collisions had high temporal locality and the respective 16-byte blocks only appeared exactly two times, so while there is some leakage, its exploitability is limited. With CIPHERFIX-ENHANCED, all collisions disappeared. All observed collisions in the analyzed targets were for 1-byte writes, which suggests that restricting CIPHERFIX-ENHANCED to 1-byte writes (and possibly 2-byte writes) is sufficient. We encountered almost no 2-byte writes in our experiments. We further discuss the security impact of the collisions in CIPHERFIX-FAST in Section 6.4.

We did not see any relevant difference between the particular PRNGs: The number of collisions is roughly equal, and there was no 32-bit mask collision even for the targets with the highest number of instrumented writes. This suggests that they are all generally suited for generating masks for the evaluated primitives within the given constraints. Nevertheless, the decision for a particular PRNG should not be made easily, as is discussed in the next section.

6.4 Balancing Security and Performance

Each variant and PRNG comes with its own advantages and drawbacks. We point out some guidelines for choosing the best composition for a given use case.

6.4.1 Properties of the implementation

To determine the most suitable variant of CIPHERFIX, one should look at the properties of the given implementation. For example, symmetric primitives, which showed a huge amount of 1-byte writes in our evaluation, do not necessarily need to be hardened against ciphertext side-channels. With hardware extensions like AES-NI and CLMUL, we found that leakage is mostly restricted to copying of inputs and outputs between encryption rounds. Thus, if the same buffer is reused for multiple blocks, the attacker may occasionally learn that a particular plaintext block or parts of it repeat. Whether this is tolerable depends on the specific use case.

6.4.2 Choosing a CIPHERFIX variant

While CIPHERFIX-FAST has the least performance overhead, it has the additional risk of leaking whether the mask and the plaintext are equal, as described in Section 6.3.1. While we did not observe that particular scenario, we saw several 1-byte collisions in `WolfSSL`'s X25519 `cswap` implementation. This suggests that CIPHERFIX-FAST and CIPHERFIX-BASE are dangerous even for algorithms with a very small number of 1-byte writes. Future work may develop a further variant that uses a merged mask/secrecy buffer but widens small writes to 4 bytes, to get both the performance benefit of CIPHERFIX-FAST and the protection of CIPHERFIX-ENHANCED. For deciding between CIPHERFIX-BASE and CIPHERFIX-ENHANCED, we generally recommend choosing the latter due to the better protection of 1-byte writes. While we observed a higher performance overhead, the difference was almost exclusively caused by the symmetric primitives which do a lot of 1-byte operations. Excluding the symmetric implementations from the geometric mean yields an overhead of 5.2x for CIPHERFIX-ENHANCED versus 4.9x for CIPHERFIX-BASE and the XorShift128+ PRNG.

6.4.3 Choosing a PRNG

Despite the high security guarantees, the considerable performance overhead of CIPHERFIX with `rdrand` suggests that this PRNG is not suitable for use with primitives that have a lot of vulnerable memory accesses. On the other hand, our custom `AES` PRNG is very fast and did not exhibit more collisions than the other PRNGs in our experiments, but is not well examined in terms of statistical properties and cycle length. Thus, as a compromise, we suggest using a fast PRNG that is well-analyzed and meets the criteria outlined in Section 6.3.1, such as XorShift128+, which only introduced a slightly higher overhead than `AES`. As a workaround for an insufficient cycle length or concerns that a high number of samples may expose weaknesses, the PRNG may be periodically reseeded with fresh entropy via instructions like `rdseed`, e.g., each time before the hardened primitive is executed. Finally, a production-level implementation of CIPHERFIX may combine different PRNGs, like a fast one for hot code paths and `rdrand` elsewhere.

6.4.4 Practical impact of overhead

Note that we focused our performance analysis on isolated cryptographic primitives, which does not reflect their typical use case. Instead, they are usually embedded into a higher-level application like a network protocol, which limits the practical influence of a moderate overhead in a specific component. For example, in TLS, only the handshake is subject to

asymmetric cryptography that needs to be hardened against ciphertext side-channel attacks. The predominant part of the protocol's runtime, the symmetric encryption and transmission of the payload, may not need as much costly protection.

7 Discussion

We conclude our study with a discussion of some design decisions of CIPHERFIX, and point out possible angles for future work which may improve accuracy and performance.

7.1 Source Code vs. Binary Instrumentation

Instead of instrumenting binaries, the implementations could be hardened during compilation: As the compiler can freely adapt the code layout and is not restricted during register allocation, it can generate more efficient binaries. However, this comes with some obstacles. First, a source-based approach would need to be able to deal with handwritten assembly code, which is abundant in highly-optimized libraries like OpenSSL or libc. This assembly code is opaque to the compiler, but can be handled transparently by binary instrumentation.

A second obstacle is a leakage analysis that spans multiple libraries. At the beginning, the application developer would need to checkout the source code of all relevant dependencies, such that they can be recompiled with the appropriate protection. The compiler can then conduct a static data flow analysis that identifies all program points that may come in contact with secret data [8]. As we found during our experiments, a particular library may call a function in another library with secret parameters, so conducting a leakage analysis on a library in isolation is insufficient. This leaves two options: First, the leakage analysis can choose to protect the parameters of the entire outward facing API of a given library, such that all incoming function calls are assumed as passing secret data. However, this significant overapproximation is likely to neutralize the performance benefit of a compiler-based solution. Thus, as a second option, we may try to conduct the leakage analysis over all code bases at once. This is hindered by the fact that static analysis of a large code base like OpenSSL or libc is already difficult, and even more so when looking at several such code bases with different build systems and structure. At the very least, it would require lots of manual tuning by the application developer.

An alternative to binary rewriting that is worth exploring for a production-level implementation of CIPHERFIX is a hybrid approach combining dynamic analysis and compiler-based instrumentation: First, a dynamic analysis is conducted over all libraries as described in Section 4. However, the results are then not used to instrument the binaries using SBI, but are sent back to the compiler. A suitable level for this is the intermediate representation (IR) of LLVM: The IR can be executed through a VM, enabling dynamic analysis. At the same time, it is abstract enough to still allow compiler optimizations between inserting the masking code and generating ELF binaries. Applying the analysis and instrumentation to IR also avoids the practical problems of dealing with large code bases, as those can be normally translated and linked into IR files. However, contrary to binary rewriting, this method still requires some effort from the library developer, and cannot straightforwardly deal with handwritten assembly code, that would need to be lifted to an equivalent IR representation first. Finally, advanced binary rewriting engines that generate symbolized reassemblable disassembly already offer performance similar to the compiler.

7.2 Analysis Coverage

Independent of the approach on instrumentation, we need to find all loads and stores that ever deal with protected data. Missing instructions during the secrecy analysis may lead to loading or storing invalid data, which can in turn cause functional incorrectness or crashes of the hardened binary. In constant-time implementations, there are no secret-dependent branches and memory accesses. However, it is useful to support some secret-independent control flow variation, e.g., for error handling or processing messages of varying length. As our analysis is dynamic, we have to rely on our inputs generating sufficient coverage, that is covering every possible execution path between classification and declassification of secrets. The secret tracking must not underapproximate (undertaint), as this may lead to missing leakages or instability due to instructions that cannot handle masked data. Overapproximation (overtainting) is acceptable to speed up leakage analysis, but may lead to unnecessary instrumentation and thus a higher runtime overhead. We found that few random inputs were sufficient to get the coverage needed for our analysis; however, one could also employ techniques like fuzzing to maximize the chances of finding all relevant code paths, especially when applying CIPHERFIX to non constant-time code. Fuzzing and a larger test case body would only impact the overhead of the offline analysis step.

Another approach for achieving full coverage is using a purely static analysis, which may be conducted either on binaries or as part of a pure compiler-based solution. However, even for the smaller exploitable primitives, we measured several tens of millions of executed instructions for a single dynamic analysis iteration, which poses a huge amount of instructions to analyze for a static analysis. To make this feasible, the static analysis would need to make some approximations, which would in turn increase the runtime overhead of the mitigation.

7.3 Alternatives to Masking

Our masking approach ensures that the written values are independent from the actual plaintexts. However, as mentioned

in [31], instead of randomizing the values written to the same address, it is also possible to randomize the address itself. This approach would need a separate memory area for secret data. The area can, for example, be implemented as a queue with used and free space that is updated with each write. The original memory locations then point to the corresponding block in the secure memory area. The resulting memory overhead becomes a security parameter: The bigger the secure memory area, the lower the risk of collisions. In early experiments, we found that the instrumentation for this approach would have significantly higher overhead due to the management of the queue. It is better suited for narrow cases where code that deals with a well-defined data structure is hardened manually, e.g., the register save/restore during a kernel context switch. In our setting, we do not see an advantage of using randomized addresses instead of masking.

In a compiler-based setting, it is also possible to securely store data by interleaving it with random nonces. For example, each 16-byte block in AMD SEV can be split into two 8-byte halves, where the first half receives the payload, while the second half is treated as a nonce that is incremented on each write. Note that this has to be done in a single step, so the entire block may need to be buffered in a vector register, that is then written at once. This method guarantees that there are no collisions for 2^{64} writes to a given block, and has a higher locality of memory accesses, as no mask buffer is necessary. In addition, reads are almost as fast as for unprotected data, as no decoding is necessary. However, it has a high implementation complexity, as the compiler has to detect code that uses pointers to iterate over arrays and adjust such loops accordingly. Finally, the compiler needs to install logic for detecting unaligned accesses that may span multiple payload blocks, introducing a different kind of overhead. Nevertheless, interleaving may be worth exploring for programming languages that abstract away the memory layout of data structures and do not allow raw pointers.

7.4 Compatibility to CFI

Along with constant-time code and ciphertext side-channel mitigations, there are further mechanisms for ensuring secure code execution, an important one being control flow integrity (CFI) protection. For example, Intel and AMD provide the so-called control flow enforcement technology (CET), that detects unwanted control flow modifications through a shadow stack and by enforcing that indirect jumps and calls point to special `endbr64` instructions. Besides inserting direct jumps to the instrumentation section, which may be avoided by using a more sophisticated binary rewriting framework, our ciphertext side-channel mitigation does not modify the control flow. Indirect branches still point to `endbr64` instructions, and the call stack is left untouched. Thus, CIPHERFIX is compatible with CFI mechanisms like CET.

8 Related Work

Dynamic taint analysis is a software analysis technique that is implemented in a variety of tools [15, 16, 18, 25, 27, 42]. Data flow based information tracking can support finding vulnerabilities in source or binary code. On the one hand, it can be used to increase the branch coverage of fuzzers like the AngoraFuzzer [14] or VUzzer [44] by checking on which bytes of secret inputs branching decisions are based. On the other hand, taint analysis can help to keep sensitive data always encrypted in memory through data protection tools like DynPTA [41] which is a compiler-based approach.

Automated analysis of side-channels in binaries focuses on finding non-constant-time behavior by analyzing leakages that can be modeled in different ways. There is a number of tools, which use DBI to observe leakages at runtime [51, 52] or detect secret-dependent accesses through symbolic execution [17, 49, 50]. Those existing tools for finding side-channel leakages do not cover the ciphertext side-channel attack vector, as it is not originated from a deviation in the behavior of memory accesses, but rather from the content of write accesses which affects the ciphertexts. However, they can be used to initially verify whether the code is constant-time, as non-constant-time code is even easier to attack than through the ciphertext side-channel.

Memory protection mechanisms implement the protection of sensitive data in memory. *Data space randomization* (DSR) [7] randomizes the representation of data that is stored in memory, with the aim of thwarting control flow hijacking attacks. This is done by instrumenting the code so that masks are added to or removed from variables before or after memory load and store operations. *CoDaRR* [43] extends DSR with a protection against leaking the masks that are used for DSR so that rerandomization prevents from recovering the secrets through attacking masks. These solutions are source code-based and thus not applicable for our tool.

Static binary instrumentation builds the basis for binary-level analysis and protection tools with different ways to insert additional code. The trampoline SBI approach is used by tools like *Detours* [24] and *PEBIL* [29] which relocate functions to newly-added `.text` and `.data` sections together with a redirection to these sections through 5-byte jumps. The technique is extended with inserting `int3` when a jump instruction does not fit in *BIRD* [38] and short 2-byte intermediate jumps in *DynInst* [11, 23]. In our work, we implemented an optimized combination of different jumps and `int3` to build a lightweight static instrumentation. For a production-level implementation of CIPHERFIX, a sophisticated instrumentation framework should be used, but for our study, a custom tool tailored to the interaction with the dynamic analyses was easier integrated. Another way of coping with 5-byte jumps is *instruction punning*, as implemented in *LiteInst* [13] and *E9PATCH* [21]. This technique uses address offset bytes in a jump instruction to also encode instructions, so fewer bytes

need to be overwritten. For our mitigation implementation, we did not employ instruction punning, as it introduces additional complexity and memory overhead due to the jump targets being scattered over a large memory area. *RetroWrite* [19] uses symbolization to generate reassemblable assembly that can be equipped with instrumentation passes and yields an optimized instrumented binary. Layout-agnostic binary rewriting can be performed with *Egalito* [54] that uses metadata to lift the program into a specialized intermediate representation. These approaches yield more efficient binaries, but need additional support for stripped binaries and some forms of inline assembly as used by libraries like OpenSSL, respectively.

9 Conclusion

In this work, we have presented a drop-in technique for automatically protecting binaries from leaking processed secrets through a ciphertext side-channel. Our approach comprises finding vulnerable code parts and then protecting them by preventing observable ciphertext changes based on secret data. The leakage localization technique combines dynamic binary instrumentation and dynamic taint analysis to protect only those memory accesses that deal with secrets or secret-derived data. The mitigation introduces randomness such that the plaintexts written to memory change for each write, leading to corresponding unique ciphertexts. We have shown that the highest security level of our proof-of-concept implementation can detect and mitigate all leaking memory accesses, with a very small probability of remaining leakage. Since there is no indication of fixes for existing or upcoming hardware, CIPHERFIX is a suitable approach for protecting software against the ciphertext side-channel.

Acknowledgements

We would like to thank Gregor Leander for a helpful discussion on the security of fast PRNGs, and the anonymous reviewers and our shepherd for their detailed comments and suggestions for improvement. This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under grants 427774779 and 439797619, and by Bundesministerium für Bildung und Forschung (BMBF) through the ENCOPIA and SASVI projects.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium, USENIX Security*, 2016.

[2] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, 2020.

[3] AMD. AMD Secure Encryption Virtualization (SEV) Information Disclosure, August 2021.

[4] Dennis Andriesse. *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*. No Starch Press, 2018.

[5] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[6] Daniel J Bernstein. Cache-Timing Attacks on AES, 2005.

[7] Sandeep Bhatkar and R. Sekar. Data Space Randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment, 5th International Conference, DIMVA*, 2008.

[8] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[9] Derek Bruening, Timothy Garnett, and Saman P. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO)*, 2003.

[10] David Brumley and Dan Boneh. Remote Timing Attacks are Practical. *Comput. Networks*, 2005.

[11] Bryan Roger Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.*, 2000.

[12] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. Fault Attacks on Encrypted General Purpose Compute Platforms. In *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY*, 2017.

[13] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. Instruction Punning: Lightweight Instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2017.

[14] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy, S&P 2018*, 2018.

[15] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. One Engine To Serve 'em All: Inferring Taint Rules Without Architectural Semantics. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[16] James A. Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, 2007.

[17] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. BINSEC/REL: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Transactions on Privacy and Security*, 2022.

[18] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. DECAF++: Elastic Whole-System Dynamic Taint Analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID*, 2019.

[19] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *2020 IEEE Symposium on Security and Privacy, S&P*, 2020.

[20] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure Encrypted Virtualization is Unsecure. *arXiv preprint arXiv:1712.05090*, 2017.

[21] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary Rewriting without Control Flow Recovery. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI*, 2020.

[22] Hiroshi Haramoto, Makoto Matsumoto, and Mutsuo Saito. Unveiling Patterns in XorShift128+ Pseudorandom Number Generators. *Journal of Computational and Applied Mathmatics*, 2022.

[23] Jeffrey K. Hollingsworth, Barton P. Miller, M. J. R. Goncalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. In *Proceedings of the 1997 Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1997.

[24] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *3rd Usenix Windows NT Symposium*, 1999.

[25] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2011.

[26] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption, 2021.

[27] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE*, 2012.

[28] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference*, 1996.

[29] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. PEBIL: Efficient Static Binary Instrumentation for Linux. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2010.

[30] Pierre L'Ecuyer and Richard J. Simard. TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Transactions on Mathmatical Software*, 2007.

[31] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (S&P)*, 2022.

[32] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium, USENIX Security*, 2019.

[33] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-Time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium, USENIX Security*, 2021.

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy, S&P*, 2015.

[35] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[36] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *29th USENIX Security Symposium, USENIX Security*, 2020.

[37] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys*, 2018.

[38] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. BIRD: Binary Interpretation using Runtime Disassembly. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2006.

[39] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference*, 2006.

[41] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection. In *42nd IEEE Symposium on Security and Privacy, S&P*, 2021.

[42] Feng Qin, Cheng Wang, Zhenmin Li, Ho-Seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39)*, 2006.

[43] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. CoDaRR: Continuous Data Space Randomization against Data-Only Attacks. In *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security*, 2020.

[44] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[45] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security*, 2004.

[46] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy, S&P*, 2010.

[47] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. Util::Lookup: Exploiting Key Decoding in Cryptographic Libraries. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[48] Sebastiano Vigna. Further Scramblings of Marsaglia's XorShift Generators. *Journal of Computational and Applied Mathmatics*, 2017.

[49] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *28th USENIX Security Symposium, USENIX Security*, 2019.

[50] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *26th USENIX Security Symposium, USENIX Security*, 2017.

[51] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *27th USENIX Security Symposium, USENIX Security*, 2018.

[52] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2022.

[53] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *2020 IEEE Symposium on Security and Privacy, S&P*, 2020.

[54] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-Agnostic Binary Recompilation. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*, 2020.

A Static Instrumentation Example

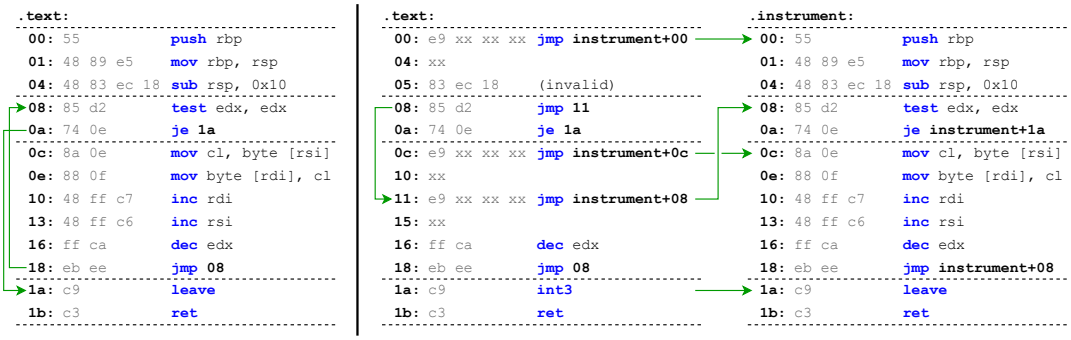

Figure 8: A simple `memcpy` implementation (left), and the resulting static instrumentation (right). The basic blocks of the original code are separated by dashed lines, control flow edges are marked with arrows. The first basic block has sufficient space for a direct 5-byte jump to the instrumentation code. The second basic block only has 4 bytes, but the third basic block offers space for two 5-byte jumps, so the second basic block gets a 2-byte jump to the third basic block (offset `11`) and from there a 5-byte jump to the instrumentation code. For the fourth basic block, all remaining space in the other basic blocks is already consumed, so it has to use an `int3` instruction. Execution that ends up at the beginning of any of the original basic blocks is always redirected to their counterparts in the `.instrument` section.

B Evaluation Results

Table 3: Memory accesses that have to be instrumented. Writes are split by their size, whereby #*n* denotes the number of *n*-byte writes. % instr. reads/writes shows the respective total percentage of instrumented accesses. Each target was iterated 10 times.

Target	# reads	instr. reads		# writes	instrumented writes						
		#	%		#1	#2	#4	#8	#16	#32	%
libsodium											
EdDSA	648,453	448,415	**69**	441,736	4,681	0	0	372,600	6,180	1,160	**87**
SHA512	200,328	82,722	**41**	104,000	810	0	0	58,718	4,800	784	**62**
mbedTLS											
AES	1,887,551	1,403,255	**74**	1,237,457	812,120	0	42	20,715	30,256	304	**70**
Base64	195,458	16,020	**8**	128,552	23,599	0	0	5,130	0	0	**22**
CC20	1,737,111	1,487,956	**86**	1,105,221	641,280	0	250,910	217	60,140	10,068	**87**
ECDH	37,328,410	3,454,726	**9**	18,773,246	0	0	881,397	1,566,188	0	1,172,058	**19**
ECDSA	7,120,602	3,301,437	**46**	3,748,086	14,240	10	260,673	1,447,753	7,674	123,806	**49**
RSA	21,203,381	12,012,577	**57**	12,068,011	1,950	10	360,804	7,303,398	1,243	122,320	**65**
OpenSSL											
ECDH	4,799,917	390,344	**8**	2,532,111	2,750	0	2,550	248,691	62	470	**10**
ECDSA	12,041,083	5,463,996	**45**	6,950,318	2,329	0	524,025	2,671,708	1,492	3,762	**46**
WolfSSL											
AES	3,661,782	1,550,484	**42**	288,454	13,150	0	90,630	60,427	10,234	0	**60**
CC20	2,603,432	1,547,406	**59**	994,267	320,320	0	476,140	25,893	20,020	0	**85**
ECDH	2,317,955	1,753,953	**76**	1,916,549	1,248	0	10,752	1,409,475	20	0	**74**
ECDSA	19,969,154	11,606,148	**58**	9,519,250	721	0	543,354	5,292,431	258,140	1,584	**64**
EdDSA	1,213,466	694,483	**57**	884,122	4,711	0	11,560	568,368	40	82	**66**
RSA	2,350,077	1,886,260	**80**	1,193,204	1,351	0	106,801	753,096	20,580	46,176	**78**