COPYCAT: Controlled Instruction-Level Attacks on Enclaves

Daniel Moghimi[1], Jo Van Bulck[2], Nadia Heninger[3], Frank Piessens[2], and Berk Sunar[1]

[1]Worcester Polytechnic Institute, Worcester, MA, USA
[2]imec-DistriNet, KU Leuven, Leuven, Belgium
[3]University of California, San Diego, CA, USA

Abstract

The adversarial model presented by trusted execution environments (TEEs) has prompted researchers to investigate unusual attack vectors. One particularly powerful class of *controlled-channel* attacks abuses page-table modifications to reliably track enclave memory accesses at a page-level granularity. In contrast to noisy microarchitectural timing leakage, this line of deterministic controlled-channel attacks abuses indispensable architectural interfaces and hence cannot be mitigated by tweaking microarchitectural resources.

We propose an innovative controlled-channel attack, named COPYCAT, that deterministically counts the number of instructions executed *within* a single enclave code page. We show that combining the instruction counts harvested by COPYCAT with traditional, coarse-grained page-level leakage allows the accurate reconstruction of enclave control flow at a *maximal* instruction-level granularity. COPYCAT can identify intra-page and intra-cache line branch decisions that ultimately may only differ in a single instruction, underscoring that even extremely subtle control flow deviations can be deterministically leaked from secure enclaves. We demonstrate the improved resolution and practicality of COPYCAT on Intel SGX in an extensive study of single-trace and deterministic attacks against cryptographic implementations, and give novel algorithmic attacks to perform single-trace key extraction that exploit subtle vulnerabilities in the latest versions of widely-used cryptographic libraries. Our findings highlight the importance of stricter verification of cryptographic implementations, especially in the context of TEEs.

1 Introduction

In the past years, we have seen a continuous stream of software-based side-channel attacks [3, 7, 24, 28, 46, 82, 83]. A first category of microarchitectural timing attacks commonly abuses optimizations in modern processors, where secret-dependent state is accumulated in various microarchitectural buffers during the victim's execution. If these buffers are not flushed before a context switch to an attacker domain, victim secrets can be reconstructed by observing timing variations by the attacker. The success of these attacks critically relies on subtle timing differences, making them inherently non-deterministic and prone to measurement noise [28]. Usually, this class of stateful attacks can be eliminated by isolating leaky microarchitectural resources [23, 45, 66, 79].

Orthogonal to the first class of microarchitectural timing attacks, recent research on *controlled-channel* attacks [29, 72, 74, 80] has abused the processor's privileged software interface to extract fully deterministic, noise-free side-channel access patterns from enclave applications. While the operating system (OS) was traditionally not considered to be under the attacker's control, this assumption fundamentally changed with the rise of trusted execution environments (TEEs), such as Intel SGX. Prior research [72, 80] has identified page-table accesses and faults as privileged interfaces that can be exploited as no-noise controlled channels to deterministically reveal enclave memory accesses at a 4 KiB page-level granularity. The paging channel has drawn considerable research attention since it abuses an intrinsic property of the x86 processor architecture without relying on microarchitectural states. In particular, controlled-channel attacks have proven to be challenging to mitigate in a principled way, in spite of numerous defense proposals [20, 21, 50, 56, 60, 61, 65].

In this work, we show that the resolution of deterministic controlled-channel attacks extends well beyond the relatively coarse-grained 4 KiB page-level granularity. We introduce COPYCAT, an innovative *interrupt-counting* channel that can precisely reconstruct the intra-page control flow of a secure enclave at a maximal, instruction-level granularity. Our attack leverages the SGX-Step [70] framework to forcibly step into a victim enclave code exactly one instruction at a time. While high-frequency timer interrupts have previously been leveraged to boost microarchitectural timing attacks [30, 34, 44, 47, 71], we exploit the architectural interrupt interface itself as a deterministic controlled channel. In short, our attacks rely on the key observation that merely counting the number of times a victim enclave can be inter-

rupted directly reveals the number of executed instructions. We show that combining our fine-grained interrupt-based counting technique with traditional, coarse-grained page-table access patterns [72, 74] as a secondary oracle allows us to construct highly effective and deterministic attacks that track enclave control flow at a maximal, instruction-level granularity. Crucially, the improved temporal dimension of COPYCAT overcomes the spatial resolution limitation of prior controlled-channel attacks, invalidating a key assumption in some prior defenses [39, 61] that presumes that adversaries can only deterministically monitor enclave memory accesses at a coarse-grained 4 KiB granularity. Furthermore, in contrast to previous high-resolution SGX side channels [3, 44, 46, 47, 71] that rely on timing differences from contention in some shared microarchitectural state, COPYCAT cannot be transparently mitigated by isolating microarchitectural resources.

To demonstrate the strength of COPYCAT, we develop single-trace attacks that allow efficient cryptographic key recovery from multiple widely-used cryptographic libraries. We extend the cryptanalysis of the binary Euclidean algorithm, which is used for modular inversion in most of the common libraries we examined, and give novel algorithms for efficiently recovering cryptographic keys from a single control flow trace for DSA and ECDSA digital signature generation and RSA key generation. The libraries we examined implemented numerous mitigations against side-channel attacks, including always-add-and-double for elliptic curve scalar multiplication and RSA exponent masking, but these protections were insufficient to protect against COPYCAT. We conclude that new classes of defenses will be necessary to protect against this type of high-granularity, deterministic, and noise-free attack.

Contributions. In summary, our main contributions are:

- We propose COPYCAT: a novel deterministic controlled-channel attack to leak runtime control flow from Intel SGX enclaves without noise at an instruction-level granularity.

- We explore the impact of COPYCAT on non-crypto applications by defeating a state-of-the-art compiler hardening technique against branch shadowing attacks.

- In an extensive empirical case study of side-channel vulnerabilities in widely-used cryptographic libraries including WolfSSL, Libgcrypt, OpenSSL, and Intel IPP, we verify the practicality and capability of these attacks, demonstrate several attacks, and report vulnerabilities in some of these libraries.

- We devise new algorithmic techniques to exploit these vulnerabilities in DSA, ECDSA, and ElGamal, as well as RSA key generation, which result in complete key recovery in the context of Intel SGX.

- Finally, we outline requirements and pitfalls for countermeasures and mitigations in hardware and software.

Responsible Disclosure. We reported the weaknesses in WolfSSL in Nov. 2019 and provided guidelines for mitigation,

tracked via CVEs 2019-1996{0,1,3} and CVE-2020-7960. We reported our findings to OpenSSL and Libgcrypt teams in Feb. 2020. OpenSSL replaced `BN_gcd` with a constant-time implementation [10] in version 1.1.1e. Libgcrypt issued a similar fix that will appear in version 1.8.6.

We shared our attack with the Intel product security incident response team (iPSIRT), who acknowledged that COPYCAT leaks side-channel information, but re-iterated that protecting against side channels requires the enclave developer to follow the constant-time coding best practices as advised by Intel [41]. Section 7 elaborates further on mitigations and explains how fully preventing our attacks requires the meticulous application of constant-time programming paradigms.

2 Background and Related Work

2.1 Side-Channel Attacks on Intel SGX

Recent Intel processors include software guard extensions (SGX) [40] to allow trusted execution of critical code in so-called *enclaves* on top of a potentially compromised OS. SGX enclaves are isolated at runtime in a memory area that is transparently encrypted and can be remotely attested by the processor. Dedicated `eenter` and `eexit` instructions switch the processor in and out of "enclave mode".

Importantly, while the confidentiality and integrity of enclaved execution is always safeguarded by the processor, traditionally privileged OS software remains in charge of availability concerns. SGX enclaves live in the virtual address space of a conventional, user-space process. To allow for demand-paging and oversubscription of the physically available encrypted memory, enclave page-table mappings are verified but remain under the explicit control of the untrusted OS. Recent address translations may be cached in an internal translation lookaside buffer (TLB), which is flushed by the processor on every enclave transition. When delivering asynchronous interrupts or exceptions, the processor takes care to securely save and scrub CPU registers before exiting the enclave, which can be subsequently re-entered through the `eresume` instruction. Furthermore, in case of a page-fault event, the processor clears the lower bits representing the page offset in the reported address to ensure that the OS can only observe enclave memory accesses at a 4 KiB page-level granularity.

While Intel SGX provides strong architectural isolation, several studies have highlighted that enclave secrets may still leak through side-channel analysis. Table 1 summarizes how all previously demonstrated side-channel attacks fall into two categories:[1] *(i)* microarchitectural timing attacks, which may achieve a high granularity but are inherently prone to measurement noise, and *(ii)* fully deterministic controlled-channel attacks that only offer a relatively coarse grained 4 KiB

---

[1] Transient-execution attacks [57, 67, 68] are orthogonal to metadata leakage through side channels and require recovery of the trusted computing base through complementary microcode and compiler mitigations.

Table 1: Characterization of demonstrated Intel SGX microarchitectural side channels (top) and controlled channels (bottom). Our novel COPYCAT technique is highlighted at the bottom and combines noise-free interrupt counting measurements with deterministic page table accesses to reconstruct enclave-private control flow at a maximal, instruction-level granularity.

	Attack	Code/Data	Granularity	Noise
μ-arch contention	DRAM row buffer conflicts [74]	Code + data	✗ Low (1-8 KiB)	✗ High
	PRIME+PROBE cache conflicts [15, 30, 47, 58]	Code + data	✗ Med (64-512 B cache line/set)	∼ Med
	Read-after-write false dependencies [46]	Data	✔ High (4 B)	✗ High
	Branch prediction history buffers [24, 34, 44]	Code	✔ High (branch instruction)	∼ Low
	Interrupt latency [71]	Code + data	✔ High (instruction latency class)	✗ High
	Port contention [3]	Code	✔ High (μ-op execution port)	✗ High
Ctrl channel	Page faults [80] and page table A/D bits [72, 74]	Code + data	✗ Low (4 KiB)	✔ Deterministic
	IA-32 segmentation faults [29]	Code + data	✗ Low/high (4 KiB; 1 B for enclaves $\leq$ 1 MiB)	✔ Deterministic
	Page table FLUSH+RELOAD [72]	Code + data	✗ Low (32 KiB)	∼ Low
	COPYCAT	**Code**	✔ **High (instruction)**	✔ **Deterministic**

page-level granularity. COPYCAT proposes the only generally applicable controlled-channel attack that is both fully deterministic and offers a maximal, instruction-level granularity.

Microarchitectural Contention. Microarchitectural timing side-channel attacks exploit the fact that various resources, such as caches [15, 30, 47, 58], DRAM row buffers [74], branch predictors [24, 34, 44], dependency resolution logic [46], or execution ports [3] are competitively shared between sibling CPU threads or not flushed when exiting the enclave. This contention causes measurable timing differences in the attacker domain, allowing the attacker to infer the private control flow or data access pattern of the enclave with varying degrees of granularity. In the context of a TEE such as Intel SGX, such attacks can be mounted with less noise and improved resolution because the adversary controls the OS.

In particular, one line of work has developed interrupt-driven attacks [30, 44, 47, 70] that rely on frequent enclave preemption to sample side-channel measurements at an improved temporal resolution. This technique has been demonstrated to amplify side-channel leakage from the cache [47], the branch target buffer [44], and the directional branch predictor [34]. Similar techniques have been applied to attack ARM TrustZone [54]. Nemesis [71] showed that while single stepping, the response time to service an interrupt may reveal which instruction is being executed in the pipeline. The SGX-Step framework [70] has been leveraged in several other microarchitectural attacks [2, 34, 57, 67, 68, 71] to reliably single-step enclaves at a *maximal* temporal resolution by means of precise and short timer interrupt intervals.

Controlled-Channel Attacks. Xu et al. [80] first showed how privileged adversaries can revoke access rights on a specific enclave page and be deterministically notified by means of a page-fault signal when the enclave next accesses that page. They demonstrated several attacks on non-cryptographic ap-

plications by observing that page-fault sequences uniquely identify specific points in the victim's execution. Subsequent work [72, 74] developed stealthier techniques to extract the same information without provoking page faults. These attacks interrupt the victim enclave to forcefully flush the TLB and provoke page-table walks, which can later be reconstructed through "accessed" and "dirty" attributes or cache timing differences for untrusted page-table entries. Finally, Gyselinck et al. [29] demonstrated an alternative controlled-channel attack that abuses legacy IA32 segmentation faults. Their attack offers an improved, byte-level granularity in the first MiB of the enclave address space, but only for the unusual case of a 32-bit enclave, and this behavior has since been fixed in recent microcode.

With COPYCAT, we contribute an improved attack technique to refine the resolution of existing controlled channels by precisely counting the number of executed enclave instructions between successive page accesses. Prior work has similarly suggested an additional temporal dimension for the paging channel by using interrupts to reconstruct `strlen` loop iterations [69, 70], or by logging noisy wall-clock time [74] for page-access events to improve stealthiness and reduce the number of TLB flushes. Recent work [42] on enclave control flow obfuscation furthermore investigated using single-stepping in an SGX simulator to probabilistically identify software versions in an emulated enclave debug environment. In contrast to these specialized cases, COPYCAT explicitly recognizes instruction counting as a practical and generically applicable attack primitive that can deterministically capture the execution trace within a single enclave code page.

2.2 Cryptographic Signature Schemes

Signature schemes are extensively used for remote attestation and authentication of trusted enclaves such as Intel SGX [39]. Moreover, TEEs like Intel SGX can promise trusted execution of these algorithms for a wide range of applications such as trusted key management [25] and private contact discov-

ery [62]. In this section, we provide an overview of signing algorithms based on public-key cryptography (PKC) that are used in our attack demonstrations.

RSA. RSA keys [53] are generated as follows:
1. Choose large prime numbers p and q, compute $N = pq$,
2. Compute the least common multiple $\lambda(N) = lcm(p\text{-}1, q\text{-}1)$,
3. Choose e such that $1 < e < \lambda(N)$ and $gcd(e, \lambda(N)) = 1$,
4. Compute $d = e^{-1} \bmod \lambda(N)$.

(N, e) are public and $(p, q, \lambda(N), d)$ are private. RSA implementations commonly use the Chinese remainder theorem (CRT) to reduce computation time, and generate additional private values $d_P = d \bmod (p-1)$, $d_Q = d \bmod (q-1)$, and $q_{inv} = q^{-1} \bmod p$. A signature is the value $s = h^d \bmod N$ where h is a hashed and padded message. Signature verification checks if $h \equiv s^e \bmod N$. To prevent side-channel attacks on signature generation, most implementations blind the input h with a random r before computing the modular exponentiation: $s_b = (hr^e)^d \bmod N = h^d r \bmod N$. Later, the unblinded signature can be computed as $s = s_b r^{-1} \bmod N$. As a result, attacks on RSA key generation have gained recent attention [2, 5]. However, since the private key parameters are only computed once, an attack against RSA key generation must only require a single trace.

DSA and ElGamal. In the Digital Signature Algorithm (DSA) [26], the public parameters are a prime p, another prime divisor n of $p-1$, and the group generator g. The private key x is chosen randomly such that $1 < x < n-1$, and the public key is $y = g^x \pmod{p}$. To sign a message hash h:
1. Choose a random secret k such that $1 < k < n-1$,
2. Compute $r = g^k \bmod p \bmod n$,
3. Compute $s = k^{-1}(h + r \cdot x) \bmod n$.

(r, s) is the output signature pair.

In the ElGamal signature scheme, an alternative to DSA, the first signature pair r is computed similarly, but the second pair is computed as $s = k^{-1}(h - r \cdot x) \bmod (p-1)$.

ECDSA. Elliptic-curve DSA (ECDSA) is similar to DSA. The public parameters are an elliptic curve E with scalar multiplication operation $\times$, a point G on the curve, and the integer order n of G over E. The secret key d is a random integer satisfying $1 < d < n-1$, and the public key is $Q = d \times G$. Signature generation for a message hash h is as follows:
1. Choose a random secret k such that $1 < k < n-1$,
2. Compute $(x, y) = k \times G$ and $r = x \bmod n$,
3. Compute $s = k^{-1}(h + r \cdot d) \bmod n$.

(r, s) is the output signature pair.

In DSA, ECDSA and ElGamal, it is critical for k to be uniquely chosen for each signature generation and to remain secret. Exposing one instance of k for a known signature results in a simple key recovery: $d = r^{-1}(s \cdot k - h) \bmod n$. Since k is an ephemeral value, a noisy side-channel attack against k cannot reduce the sampling noise using multiple runs of the attack. However, as discussed in Section 2.3, lattice attacks can recover the signing key from partial knowledge of k for many signatures. In Section 4.2 and Section 5, we show that we can recover the entire ephemeral k deterministically in a single trace of the computation of the modular inverse $k^{-1} \bmod n$. Single-trace attacks on signature generation illustrate vulnerabilities even in scenarios where an attacker cannot trigger multiple signature generation operations or can only collect a single trace.

2.3 Side-Channel Attacks on PKC Schemes

Public-key algorithms that execute variable operations for each bit of a secret input, like the square-and-multiply algorithm for modular exponentiation, and scalar multiplication based on Montgomery ladders, are susceptible to side-channel leakage. Such algorithms have been exploited in naive attacks [81, 82, 84] where the victim is triggered many times to compensate for potential sampling noise. These attacks generally conclude with the recovery of most of the secret bits. Nowadays, most implementations have adopted constant-time algorithms like fixed-window scalar multiplication to mitigate such attacks [52].

Key Recovery using Partial Information. Key recovery from DSA and ECDSA with partial knowledge of the nonce k can be solved efficiently using lattices [13, 49]. These attacks apply to the case when a few bits are leaked about the nonce for multiple signatures, and the adversary can sample many signatures. Researchers have applied lattice-based attacks to non-constant time algorithms that leak some information about k [8, 51, 55]. Garcia et al. [27] demonstrate an attack that recovers the sequence of divisions and subtractions from the binary extended Euclidean algorithm (BEEA) for modular inversion. They observe that this sequence leaks some least significant bits of k and apply a lattice-based key recovery algorithm. In contrast, COPYCAT allows full key recovery from a single DSA signature trace, even for a compact BEEA implementation (§4.2). We generalize this attack to another vulnerable modular inverse implementation used for DSA, ECDSA, and ElGamal (§5).

Even subtle implementation flaws that leak the bit length of k are sufficient for multi-trace lattice-based key recovery [16, 22, 48]. In these cases, while the algorithm was implemented with enough care to avoid secret-dependent conditional statements, they leak the bit length by skipping the most significant zero bits of k. In Section 4.4, we exploit a countermeasure against this attack to precisely leak the nonce length, and recover the secret key using a lattice attack.

Single-Trace Attacks on RSA. Recent work has demonstrated a single-trace side-channel attack against RSA key

generation that leaks the sequence of divisions and subtractions from the BEEA during the coprimality test $gcd(e, p-1)$ [4, 75] or secret key generation $d = e^{-1} \mod \lambda(N)$ [18]. These attacks recover the secrets $(p-1)$ or $lcm(p-1, q-1)$ from this sequence when e is small enough to be brute forced, which is typically the case in practice[2]. The proposed mitigation is to increase the size of the input e by masking it with a random variable that may be hard coded [18]. In Section 4.3, we use COPYCAT to recover all the branches from BEEA, not just the sequence of divisions and subtractions. We propose a novel algorithm that uses this information to recover the private factors p and q from $e^{-1} \mod \lambda(N)$. Our attack works even for large e, thwarting the above mitigations.

Furthermore, our algorithm is even able to recover the key from a modular inversion algorithm with multiple unknowns. We demonstrate a novel end-to-end single-trace attack on the CRT computation $q^{-1} \mod p$. In a concurrent and independent work, Aldaya et al. [2] outline a different key recovery algorithm for $q^{-1} \mod p$ that is not always successful. Our single-trace attacks on RSA in Section 4.3 use a branch-and-prune algorithm inspired by Heninger and Shacham [31]. Bernstein et al. applied a variant of branch-and-prune algorithm to recover RSA keys from a sliding-window modular exponentiation implementation [9]. Similarly, Yarom et al. demonstrated an attack with intra-cache line granularity on a fixed-window implementation of modular exponentiation that recovers a fraction of the bits [83]. In Section 5, we generalize our attack to implementations of BEEA used in other popular cryptographic libraries. We demonstrate attacks against $gcd(p-1, q-1)$ in OpenSSL X.931 RSA and $q^{-1} \mod p$ and $e^{-1} \mod \lambda(N)$ in WolfSSL and Libgcrypt.

3 COPYCAT Attack

Attacker Model. We assume the standard Intel SGX root adversary model with full control over the untrusted OS [40]. SGX's strong threat model is justified, for instance, by considering untrusted cloud providers under the jurisdiction of foreign nation states, or end users with an incentive to break DRM technology running on their own device. Following prior work, we assume a remote, software-only adversary who has compromised the untrusted OS, allowing the x86 APIC timer device to be configured to precisely interrupt the enclave [30, 44, 47, 70] and modify page-table entries to learn enclaved memory accessed at a 4 KiB granularity [61, 72, 80]. Like previous attacks, we further assume knowledge of the victim application, either through source code or the application binary. We assume the enclave code is free from memory-safety vulnerabilities [69] and the Intel SGX platform is properly updated against transient-execution attacks [57, 67].

The adversary's goal is to learn fine-grained control-flow decisions in the victim enclave. In contrast to noisy microar-

chitectural side channels [3, 15, 44, 46, 47, 71], we can also target victims who process a secret only *once* in a single run (as is the case in key generation) and hence victims who cannot be forced to repeatedly perform computations on the same secret multiple times. Crucially, in contrast to prior controlled-channel attacks [72, 80], COPYCAT offers intra-page granularity and we assume that conditional control flow blocks in the victim enclave are aligned "to exist entirely within a single page" as officially recommended by Intel [39].

3.1 Building the Interrupt Primitive

Debug features like the x86 single-step trap flag are explicitly disabled by the Intel SGX design [40] while in enclave mode. Recent research, however, has demonstrated that root adversaries may abuse APIC timer interrupts to forcibly pause a victim enclave at fixed time intervals. We build our interrupt primitive on top of the open-source SGX-Step [70] framework, which offers a maximal temporal resolution by reliably interrupting the victim enclave at most one instruction at a time. SGX-Step comes in the form of a Linux kernel driver and runtime library to configure APIC timer interrupts and untrusted page-table entries directly from user space.

Deterministic Single-Stepping. We first establish a suitable value for the platform-specific `SGX_STEP_TIMER_INTERVAL` parameter using the SGX-Step benchmark tool on our target processor. This ensures that the victim enclave always executes at most one instruction at a time. Previous studies [34, 70, 71] have reported reliable single-stepping results with SGX-Step for enclaves with several hundred thousand instructions where in the vast majority of cases ($> 97\%$) the timer interrupt arrives within the *first* enclave instruction after `eresume`, *i.e.*, single-step, and in *all* other cases the interrupt arrives within `eresume` itself, *i.e.*, zero-step before an enclave instruction is ever executed. Furthermore, zero-step events can be filtered out by observing that the "accessed" bit in the untrusted page-table entry mapping the enclave code page is only ever set by the processor when the interrupt arrived after `eresume` and the enclave instruction has indeed been retired [71]. Hence, to achieve noiseless and deterministic single-stepping for revealing code and data accesses at an instruction-level granularity, we rely on the observation that a properly configured timer *never* causes a multi-step, and we discard any zero-step events by querying the "accessed" bit in the untrusted page-table entry mapping the current enclave code page. The experimental evaluation in Section 4 confirms that our single-stepping interrupt primitive indeed behaves fully deterministically when using COPYCAT to count several millions of enclave instructions.

Before entering single-stepping mode, we first use a coarse-grained page-fault state machine to easily advance the enclaved execution to a specific function invocation on the tar-

---

[2]e is commonly chosen as $2^{16} + 1 = 65537$.

geted code page. Such page-fault sequences have priorly been shown to uniquely locate specific execution points in large binaries [61, 75, 80]. Once the specific code page of interest has been located, COPYCAT starts counting instructions until detecting the next code or data page access to reveal instruction-level control flow.

Effects of Macro Fusion. Interestingly, we found that COPYCAT can also be used to study a microarchitectural optimization in recent Intel Core processors, referred to as *macro fusion* [38, 77]. The idea behind this optimization technique is to combine certain adjacent instruction pairs in the front-end into a single micro-op that executes with a single dispatch and hence frees up space in the processor pipeline.

Intel documents that fusion only takes place for some well-defined compare-and-branch instruction pairs [38, §3.4.2.2], which are additionally not split on a cache line boundary [38, §2.4.2.1]. We experimentally found that for fusible instruction pairs, COPYCAT consistently counts only *one* interrupt, even though the enclave-private program counter has been advanced with *two* assembly instructions forming the fused pair. Our experimental observations on Kaby Lake confirm Intel's documented limitations, e.g., `test;jo` can be fused (interrupted once) but not `cmp;jo` (interrupted twice); and fusible pairs that are split across an exact cache line boundary are not fused (interrupted twice). Importantly, we found that macro fusion does *not* impact the reliability of COPYCAT as a deterministic attack primitive. That is, we consistently observed in all of our attacks that macro fusion depends solely on the architectural program state, *i.e.*, opcode types and their alignments, and hence a given code path always results in the same, deterministic number of interrupts.

To the best of our knowledge, COPYCAT contributes the first methodology to independently research and reverse-engineer macro fusion optimizations in Intel processors. While our observations confirm that macro fusion behaves as specified, we consider a precise understanding of macro fusion of particular importance for compile-time hardening techniques that balance conditional code paths (§7).

3.2 Instruction-Level Page Access Traces

Leakage Model. COPYCAT complements the coarse-grained 4 KiB spatial resolution of previous page fault-driven attacks with a fully deterministic temporal dimension. By interrupting after every instruction and querying page-table "accessed" bits, COPYCAT adversaries obtain an instruction-granular trace of page visits performed by the enclave. This trace may reveal private branch decisions whenever a secret-dependent execution path does not access the exact same set of code and data pages at every instruction offset in both branches. Importantly, even when both execution paths access the same sequence of code and data pages, and hence remain indistinguishable for a traditional page-fault adversary [80],
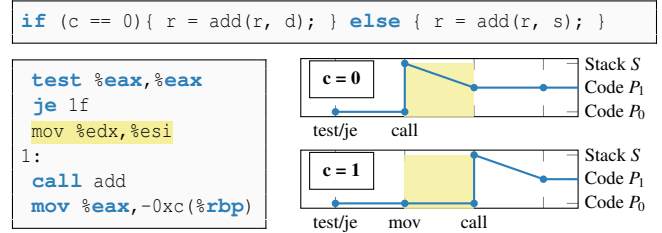

Figure 1: Balanced if/else statement (top), compiled to assembly (left). Precise page-aligned, intra-cache line conditional control flow can be deterministically reconstructed with instruction-granular COPYCAT page access traces (right).

we show below that compilers may in practice still emit unbalanced instruction counts between page accesses in both branches. Sections 6 and 7 elaborate further on the limitations of this leakage model and the precise requirements for static code balancing solutions.

If/Else Statement. Conditional branches are pervasive in all applications [30, 32, 44, 80], but even side-channel hardened cryptographic software may assume that carefully aligned if/else statements or tight loops cannot be reliably reconstructed (§4). Figure 1 provides a minimal example of an if statement that has been hardened using a balancing else branch, e.g., as in the Montgomery Ladder algorithm. The corresponding assembly code, as compiled by `gcc`, indeed only differs in a single x86 instruction that can fit entirely within the same page and cache line. This if branch is hence indistinguishable for a page-fault or cache adversary. While finer-grained, branch prediction side channels may still be able to reconstruct the branch outcome, these attacks typically require several runs of the victim and can be trivially addressed by flushing the branch predictor on an enclave exit.

Figure 1 illustrates how COPYCAT can deterministically reconstruct the branch outcome merely by counting the number of instructions executed on the P_0 code page containing the if branch before control flow is eventually transferred to the P_1 code page containing the `add` function, as revealed by probing the "accessed" bit in the corresponding page-table entry. The example furthermore highlights that even if all of the code were to fit on a single code page $P_0 = P_1$, COPYCAT adversaries could still distinguish both branches by comparing the relative position of the data access to the stack page S performed by the `call` instruction. In particular, while traditional page-fault adversaries always see the same *page fault sequence* (P_0, S, P_1), independent of the secret, COPYCAT enriches this information with precise instruction counts, resulting in distinguishable *instruction-level page access traces* (P_0, P_0, S, P_1) vs. (P_0, P_0, P_0, S, P_1).

```
switch(c)
{
  case 0:
    r = 0xbeef;
    break;
  case 1:
    r = 0xcafe;
    break;
  default:
    r = 0;
}
```
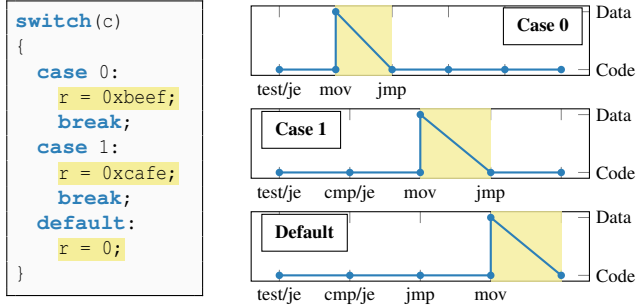
Figure 2: Conditional data assignments in a page-aligned switch statement (left) deterministically leak through their relative positions in the precise, instruction-granular page access traces extracted by COPYCAT (right).

Switch-Case Statement. As a further example, Figure 2 illustrates precise control-flow recovery in a switch-case statement, where the code blocks again fall entirely within a single page and cache line, and where the same data is accessed in every case. While traditional page-fault adversaries always observe an identical, input-independent access sequence to the code and data pages, and the tight sequence of conditional jumps poses a considerable challenge for branch prediction adversaries [44], COPYCAT deterministically reveals the entire control flow through the *relative* position of the data access in the instruction-granular page access traces.

3.3 Defeating Branch Shadowing Defenses

To highlight the importance of COPYCAT for non-cryptographic applications, we employ its improved resolution to defeat a state-of-the-art compiler defense [32] against branch predictor leakage. This again shows that COPYCAT changes the attack landscape and requires orthogonal mitigations when compared to microarchitectural side channels.

Branch Shadowing Mitigation. Lee et al. [44] first proposed Zigzagger, an automated compile-time approach to defend against branch-shadowing attacks by rewriting conditional branches using cmov and a tight trampoline sequence of unconditional jump instructions. However, the security of their compiler transformation critically relies on the trampoline sequences being non-interruptible, and several proof-of-concept attacks on Zigzagger have been demonstrated using precise interrupt capabilities [29, 70, 71]. In response, Hosseinzadeh et al. [32] designed an improved compiler mitigation that employs runtime randomization to dynamically shuffle jump blocks on the trampoline area, thereby effectively hiding branch targets and making branch shadowing attacks probabilistically infeasible. Figure 3 illustrates how conditional branches are redirected through randomized jump locations ① on the trampoline page, while ensuring that all jumps ② outside of the trampoline are always executed in the
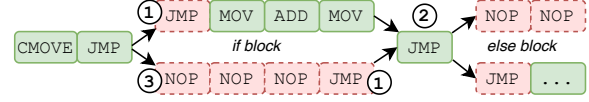

Figure 3: Compiler mitigation [32] for branch prediction side channels. COPYCAT reveals control flow via the number of instructions executed on the trampoline page (red, dashed).

same order. Finally, to protect against timing attacks, trampoline code is explicitly balanced with dummy instructions ③ to compensate for skipped blocks in the instrumented code.

Case-Study Attack. We evaluated COPYCAT on the open-source[3] release of the compiler hardening scheme [32] based on LLVM 6.0. First, we found that the dummy instruction balancing pass is not always entirely accurate and may result in execution paths that differ slightly by one or two instructions (cf. Appendix A.2). Crucially, while such subtle deviations would indeed very likely not be exploitable through timing, as originally envisioned by the mitigation, we experimentally validated that the unbalanced paths can be fully deterministically distinguished by COPYCAT adversaries. Second, even when the code paths are perfectly balanced, Figure 3 illustrates that merely counting the *number* of instructions executed on the trampoline page deterministically reveals whether the victim is executing balancing dummy code in a trampoline block or the actual if block on the instrumented code page. Note that the compiler carefully maintains a constant jump order when moving back and forth between the trampoline area and the instrumented code, ensuring that the execution remains oblivious to classical page-fault adversaries [61, 80] who will always observe the exact same *sequence* of pages regardless of the actual code blocks being executed.

4 Unleashing COPYCAT on WolfSSL

WolfSSL is a prominent, FIPS-certified solution officially supporting Intel SGX [78]. In a case study on the WolfSSL cryptographic library, we show that COPYCAT enables attacks that were not previously possible without a deterministic and fine-grained leakage model. In Section 4.1, we outline our controlled-channel attack using COPYCAT to precisely recover the full execution trace of WolfSSL's implementation of the binary extended Euclidean algorithm (BEEA), which is used for modular inversion of cryptographic secrets in DSA, ECDSA, and RSA. Precise recovery of the full execution flow of BEEA enables new single-trace algorithmic attacks on both DSA signing and RSA key generation, as demonstrated in Sections 4.2 and 4.3, respectively. Finally, we apply COPYCAT to bypass incomplete side-channel mitigations and recover de-

---

[3]Branch shadowing mitigation: https://github.com/SSGAalto/sgx-branch-shadowing-mitigation

terministic partial information on ECDSA signatures, which allows for efficient key recovery via lattices.

Experimental Setup. Our experimental setup includes a desktop Intel Core i7-7700 CPU that supports Intel SGX and is updated with the latest microcode (0xca) running Ubuntu 16.04 with kernel 4.14.0-72-generic. We use the SGX-Step [70] framework v1.4.0 to implement our attacks on the latest stable WolfSSL version 4.2.0. WolfSSL officially supports compilation for Intel SGX enclaves. We implemented our key recovery attacks in SageMath version 8.8.

4.1 COPYCAT on BEEA

Computing the modular inverse or greatest common divisor (GCD) using the binary extended Euclidean algorithm (BEEA) has previously exposed cryptographic implementations to side-channel attacks [1, 27, 75]. The BEEA, as shown in Algorithm 1, is not constant time and can leak various bits of its input. However, previous attacks are limited to recovering only partial and noisy information about the secret input. This limitation stems from low spatial resolution and the presence of noise. For instance, a cache- or page-level attacker who can distinguish which arithmetic subroutines have been invoked cannot determine the outcome of the comparison at line 13 since both directions of the branch generate exactly the same sequence of memory access patterns. In addition, the arithmetic functions may fit within the same page and become indistinguishable for a page-level adversary. Alternatively, a cache attacker may try to track the outcome of these branches within the same page by tracking the corresponding instruction cache lines for the BEEA subroutine. However, a compact implementation of this algorithm can fit multiple branches within the same cache line. While some microarchitectural attacks on the instruction stream may leak some of these low-level branch outcomes, they are all prone to various amounts of noise [3, 24, 34, 44, 71].

WolfSSL supports two different BEEA implementations in subroutines `fp_invmod_slow` and `fp_invmod`.[4] The former is a straightforward implementation, and the latter is a compact implementation that only supports odd moduli. We analyze both implementations and show how to use COPYCAT to recover the runtime control flow of these implementations deterministically and without noise.

Binary Layout of Modular Inversion. After compilation, the subroutines `fp_iseven` and `fp_isodd` are simply inlined within the same page as their caller `fp_invmod_slow`. However, the arithmetic functions A=`fp_add`, C=`fp_cmp`, D=`fp_div_2`, and S=`fp_sub` are external calls and reside in a new page. Analyzing these arithmetic functions (A, C, D,

---

[4] `fp_invmod_slow` and `fp_invmod` can be found at line 885 and 1015 of https://github.com/wolfSSL/wolfssl/blob/48c4b2fedc/wolfcrypt/src/tfm.c, respectively.

Algorithm 1 Modular inversion using the BEEA. In the optimized compact implementation when the modulus is odd, highlighted (blue) statements are removed.

1: **procedure** MODINV(u, modulus v)
2:     $b_i \leftarrow 0$ $d_i \leftarrow 1, u_i \leftarrow u, v_i = v, a_i \leftarrow 1, c_i \leftarrow 0$
3:     **while** $isEven(u_i)$ **do**
4:         $u_i \leftarrow u_i/2$
5:         **if** $isOdd(b_i)$ **then**
6:             $b_i \leftarrow b_i - u, a_i \leftarrow a_i + v$
7:         $b_i \leftarrow b_i/2, a_i \leftarrow a_i/2$
8:     **while** $isEven(v_i)$ **do**
9:         $v_i \leftarrow v_i/2$
10:         **if** $isOdd(d_i)$ **then**
11:             $d_i \leftarrow d_i - u, c_i \leftarrow c_i + v$
12:         $d_i \leftarrow d_i/2, c_i \leftarrow c_i/2$
13:     **if** $u_i > v_i$ **then**
14:         $u_i \leftarrow u_i - v_i, b_i \leftarrow b_i - d_i, a_i \leftarrow a_i - c_i$
15:     **else**
16:         $v_i \leftarrow v_i - u_i, d_i \leftarrow d_i - b_i, c_i \leftarrow c_i - a_i$
17:     **return** d_i

S), including their internal subroutines, shows that they span $2,895$ bytes. Hence, it is reasonable to assume that they can fit into a single 4 KiB page, thus preventing a page-level attacker from distinguishing them at runtime altogether. In addition, even assuming they do not align within the same page, reconstructing the exact execution flow is still impossible. For example, the transition from S to D can result from multiple different code paths. The instructions for `fp_invmod_slow` can fit into fewer than 6 cache lines with multiple basic blocks[5] overlapping within the same line.

WolfSSL also supports a modified version of BEEA, `fp_invmod` specialized to the case of odd modulus, which is used for RSA q^{-1} mod p (§4.3) and DSA k^{-1} mod n (§4.2). The control flow and overall layout for `fp_invmod` are similar to the above implementation but it is more compact, as some of the arithmetic statements have been removed. `fp_invmod` can fit into fewer than 4 cache lines with multiple overlapping basic blocks.

Recovering BEEA Control-Flow Transfers. We analyzed the runtime control flow of `fp_invmod_slow` by matching its disassembly with the execution trace we recovered from running COPYCAT. Figure 4 shows the control flow transfers at page-level granularity for the page corresponding to `fp_invmod_slow` and the page corresponding to arithmetic functions (Circles). Additionally, the weight of each arrow shows the number of instructions that are executed for `fp_invmod_slow` before accessing the page corresponding to arithmetic functions. The division loop for u_i (u-loop) and v_i (v-loop) have a similar control flow. In addition, the two blocks of substitutions after the comparison of $u > v$ have similar control flow for both the left $S1$ and right $S2$ direction.

---

[5] A basic block is a code sequence that has no branches in and out.

Figure 4: Control flow of the BEEA as implemented by `fp_invmod_slow`. Each circle (D=div, C=cmp, S=sub, A=add) represents a call to a function in the page that holds these arithmetic functions. We count the exact number of instructions between two consecutive invocations that hit this page. The instruction counts reveal branch outcomes.

Figure 5: An example cut of a trace that is recovered from `fp_invmod_slow`. First, the weights are replaced according to Rules 1, 2, and 3. Then other transitions (Rules 4 and 5) are used to recover the whole control flow sequence.

Only certain transitions are viable from these blocks to division loops during the computation of the modular inverse. For example, *S2* always goes to *v-loop* and *S1* always goes to *u-loop*. Since these instruction counts are distinguishable for transitions that are related to conditional statements, we can use a trace consisting of a vector of these weights in the graph to infer the outcome of the conditional statement.

With a trace including the weights of instruction counts collected between two consecutive accesses to the page that holds the arithmetic operations (A, C, D, S), we apply a set of divide-and-conquer rules to reconstruct the control flow for `fp_invmod_slow`. These rules start by translating the recovered weights to corresponding generic blocks. For example, every time the algorithm executes an iteration of a division loop (u/v-loop), we observe either the sequence $D \to D \to D$, or the sequence $D \to A \to S \to D \to D$. Each of these sequences generates a consistent set of weights. Similarly, *S1* or *S2* always generates a sequence like $C \to S \to S \to S$. After translating these generic blocks, we can use the remaining transitions to distinguish the exact blocks, i.e., we can recover whether a *S1* or *S2* followed by a set of division loops is equal to a transition from *S1* to *u-loop* or transition from *S2* to *v-loop*. These rules are summarized as follow:

- **Rule 1:** $? \xrightarrow{11} ? \xrightarrow{3} ? = D \to D \to D$.

Figure 6: Control flow of BEEA in `fp_invmod`.

- **Rule 2:** $? \xrightarrow{13} ? \xrightarrow{4} ? \xrightarrow{3} ? \xrightarrow{3} ? = D \to A \to S \to D \to D$.
- **Rule 3:** $? \xrightarrow{5} ? \xrightarrow{4} ? \xrightarrow{4} ? = C \to S \to S \to S$.
- **Rule 4:** $S? \xrightarrow{13} ? = S2 \to v\text{-}loop$.
- **Rule 5:** $S? \xrightarrow{8} ? = S1 \to u\text{-}loop$.

We first replace some of the weights according to Rules 1, 2, and 3, which identify if we are in a division loop (u-loop or v-loop) and a comparison and substitution block (S?). Then based on the other transitions (Rule 4 and 5), we can determine which state of the comparison and substitution block we have moved from, and which division loop we have moved to within the trace. An example sequence from the execution of *fp_invmod_slow* and its translation to the control flow transitions is given in Figure 5.

For the compact implementation in *fp_invmod*, we apply the same approach. Figure 6 shows the control flow for this implementation after runtime analysis using COPYCAT. Similarly, we define a set of rules to translate the trace of instruction counts to control flow transfers of BEEA. Based on Figure 6, we modify the first three rules as follows to support control-flow recovery based on the same approach:[6]

- **Rule 1:** $? \xrightarrow{7} ? = D \to D$.
- **Rule 2:** $? \xrightarrow{8} ? \xrightarrow{3} ? \xrightarrow{3} ? = D \to S \to D$.
- **Rule 3:** $? \xrightarrow{5} ? \xrightarrow{4} ? = C \to S \to S$.

4.2 Single-Trace Attack on DSA Signing

In contrast to previous attacks on BEEA that leak partial information about the nonce [27], COPYCAT recovers virtually the entire control flow from the execution of this implementation with 100 percent precision. As a result, we can perform a single-trace attack on the DSA signing operation. In Section 5, we generalize this attack and expose multiple vulnerabilities in the *Libgcrypt* library.

DSA Key Recovery. WolfSSL uses `fp_invmod` to compute the modular inversion of $k_{inv} = k^{-1} \bmod n$, where n is an odd prime. Since we can recover the exact control flow of this computation and the modulus n is public, we simply

---

[6]Rule 4 and 5 remain the same.

step through the execution trace of Algorithm 1, applying each step of the computation according to the recovered trace to compute k_{inv} bit by bit. After recovering k_{inv}, recovering the full nonce and private key is trivial: $k = k_{inv}^{-1} \mod n$, $x = r^{-1}(sk - h) \mod n$.

Evaluation. To attack 160-bit DSA, we used a combination of pages in a page-level controlled-channel attack to first reach the beginning of the modular inversion operation for DSA. Then we start COPYCAT over the code page for `fp_invmod`. We executed this attack for 100 different signing operations. On average, this attack issues 22,000 IRQs and takes 75 ms to iterate over an average of 6,320 steps for each signature generation. Out of 100 experiments, our single-trace attack successfully recovered the full control flow and the key using the algorithm above, implying that COPYCAT reliably reconstructs the entire execution flow. As a result, a single-trace attack on DSA can be executed without the need for multiple signatures.

4.3 Single-Trace Attacks on RSA KeyGen

During RSA key generation, WolfSSL checks if a potential prime p is coprime with e by checking if $gcd(e, p-1)$ is equal to 1. This step uses the textbook greatest common divisor (GCD) algorithm, which simply performs a series of divisions. This algorithm appears to be less vulnerable to control-flow-based key recovery. However, in a later stage, WolfSSL computes $d = e^{-1} \mod \lambda(N)$ and the CRT parameter $q^{-1} \mod p$ using the BEEA. WolfSSL always generates the CRT parameters during RSA key generation.[7]

Key Recovery from a $q^{-1} \mod p$ Trace. Compared to $k^{-1} \mod n$, this attack is more challenging since in this case, both operands p and q are unknown. We give a novel and efficient attack that recovers the private RSA parameters p and q using COPYCAT. We use the relationship of the public modulus $N = pq$ and the execution trace of the BEEA on $q^{-1} \mod p$, which provides enough information to recover the factorization of N. The main idea is that the BEEA algorithm works sequentially from the least significant bits of p and q. Thus if we iteratively guess bits of p and q starting from the least significant bits, we can verify that a guess matches the relevant steps of the BEEA execution trace, as well as the constraint that $N = pq$ for the bits guessed so far, and eliminate guesses that do not. This algorithm resembles the branch-and-prune algorithm of [31], with new constraints.

We propose Algorithm 2 to recover p and q using only knowledge of N and the execution trace of the BEEA on $q^{-1} \mod p$. The algorithm starts by initializing a list of hypotheses for values of the least significant bits of q and p. Each

---

Algorithm 2 Recovering p and q from trace of $q^{-1} \mod p$.

```
 1: procedure RECOVER_PQ(trace t, modulus N)
 2:     h ← (−test_t(t, 1, 1), 1, 1, 1)
 3:     while h do
 4:         steps, b, p, q ← hpop(h)
 5:         if p.q = N then return p, q
 6:         g ← (p, q), (p + 2^b, q), (p, q + 2^b), (p + 2^b, q + 2^b)
 7:         for p_s, q_s in g do
 8:             if mod(p_s.q_s, 2^{b+1}) = mod(N, 2^{b+1}) then
 9:                 hpush(h, (−test_t(trace, p_s, q_s), b + 1, p_s, q_s))
```

---

hypothesis keeps track of the current step, bit position b, and the hypothesized values of $p_s = p \mod 2^b$ and $q_s = q \mod 2^b$. Among the four possible assignments for the $(b+1)^{st}$ bits of p and q in Step 7, there will be two choices satisfying the constraint that $pq \equiv N \mod 2^{b+1}$. For these new guesses, we evaluate the BEEA algorithm up to the number of bits guessed so far, and check this deterministic algorithm evaluation on the guess against the ground truth execution trace t. We then do a depth-first search prioritized by the number of steps in which the algorithm executed correctly, and terminate when we have found a candidate for which $pq = N$ holds.

Evaluation. We executed an attack similar to Section 4.2 to collect traces from the modular inversion of $q^{-1} \mod p$, as it is computed by `fp_invmod_slow`. We tried this attack on 100 different 2048-bit RSA key generations. On average, we iterate over 39,400 steps by issuing 106490 IRQs in 365 ms. However, the average time to collect a trace can take up to a second depending on how fast the prime numbers are chosen. The attack takes 20 seconds to recover the key from a trace. All 100 trials of the attack successfully recovered the keys.

Key Recovery from an $e^{-1} \mod \lambda(N)$ Trace. In contrast to previous attacks on this computation [18], we propose a different algorithmic attack that takes advantage of the fact that COPYCAT can recover the entire control flow of this algorithm. As a result, a single-trace attack can be carried out for any value of e, both large or small. This shows that the proposed masking countermeasure in [18] is insecure against our strong COPYCAT adversary.

Our goal is to recover the RSA primes p and q using the trace of the BEEA for $d = e^{-1} \mod \lambda(N)$. The modulus N and the public exponent e are known, while $\lambda(N)$ is secret. We present a modified branch-and-prune technique in Algorithm 3 that recovers the factors p and q for a large fraction of generated RSA keys.

The main idea is to iteratively guess bits of p and q starting from the least significant bits, then verify that $pq = N$ and the relevant steps of the BEEA execution trace match the guess so far. However, the BEEA is computed on e and $\lambda(N) = (p-1)(q-1)/\gcd(p-1, q-1)$. We do not know $\gcd(p-1, q-1)$ and must guess it for this algorithm, but with high probability it only has small factors and can be brute forced.

---

[7] `wc_MakeRsaKey` at https://github.com/wolfSSL/wolfssl/blob/48c4b2fedc/wolfcrypt/src/rsa.c#L3726 invokes BEEA multiple times during RSA Key generation.

Algorithm 3 Recovering p and q from trace of $e^{-1} \bmod \lambda$.

1: **procedure** RECOVER_PQ(trace t, e, modulus N)
2:     $h \leftarrow (-test\_t(t,0,e), 1, 1, 1)$
3:     **while** h **do**
4:         $steps, b, p, q \leftarrow hpop(h)$
5:         **if** $p.q = N$ **then return** p, q
6:         $g \leftarrow (p,q), (p+2^b, q), (p, q+2^b), (p+2^b, q+2^b)$
7:         **for** p_s, q_s in g **do**
8:             **if** $mod(p_s.q_s, 2^{b+1}) = mod(N, 2^{b+1})$ **then**
9:                 $\phi = (p_s - 1)(q_s - 1)$
10:                 **for** $i = 1, \dots, 2^{\ell}$ **do**
11:                     **if** $p_s q_s > N$ or $mod(\phi, 2^i) \neq 0$ **then**
12:                         continue
13:                     $\lambda = \phi/2^i$
14:                     $newsteps = test\_t\_lamda(t, \lambda, e)$
15:                     **if** $newsteps >= b + 1$ **then**:
16:                       $hpush(h, (-newsteps, b+1, p_s, q_s))$
        return fail

```c
int wc_ecc_mulmod_ex(mp_int* k, ecc_point *G, ecc_point *R, mp_int* a, mp_int
        * modulus, int map, void* heap) { ...
for (;;) {
if (--bitcnt == 0) { /* grab next digit as required */
  if (digidx == -1) {
    break;
  }
  buf = get_digit(k, digidx);
  bitcnt = (int)DIGIT_BIT;
  --digidx;
}
i = (buf >> (DIGIT_BIT - 1)) & 1; /* grab the next msb from the multiplicand */
buf <<= 1;
if (mode == 0) {
  mode = i; /* timing resistant - dummy operations */
  err = ecc_projective_add_point(M[1], M[2], M[2], a, modulus, mp);...
  err = ecc_projective_dbl_point(M[2], M[3], a, modulus, mp);...
}...
err = ecc_projective_add_point(M[0], M[1], M[i^1], a, modulus, mp);...
err = ecc_projective_dbl_point(M[2], M[2], a, modulus, mp);...
} /* end for */...}
```

Listing 1: `wc_ecc_mulmod_ex` implements scalar multiplication using a bit-by-bit always-add-and-double algorithm. The function protects against both timing and cache attacks by executing dummy instructions. For brevity, error checking and code sections that are not relevant to our attack are removed.

For simplicity, we specialize to the case of $\gcd(p-1, q-1) = 2^i$ for small integer i below, but the analysis can be extended to other candidate small primes with more brute force effort.

For each guess 2^i for $\gcd(p-1, q-1)$, we iteratively generate guesses for p_s and q_s, compute $\phi_s = (p_s - 1)(q_s - 1)$ and then $\lambda_s = \phi_s/2^i$. We compare the execution trace t to the execution trace for λ_s and e. The algorithm either returns p and q or it fails to recover p and q if $\phi/\lambda(N) \neq 2^i$.

Analysis. The algorithm will succeed whenever $\phi/\lambda = 2^i$ for small i. For non-powers of 2 the test against the BEEA execution trace in Step 15 will likely fail, and cause this branch to be pruned. Since $p = 2p' + 1$ and $q = 2q' + 1$ for some $p', q' \in \mathbb{Z}$, we have $\lambda(N) = lcm(p-1, q-1) = 2lcm(p', q')$. From the prime number theorem [59], the probability that two random integers are coprime converges to $\prod_{p \in primes}(1 - 1/p^2) = \frac{6}{\pi^2} \approx 61\%$ as the size of the integers increases. In other words, if we run Algorithm 3 for only $i = 1$, it will succeed 61% of the time when p' and q' are actually coprime. If we allow p' and q' to have even factors we obtain a probability of $\prod_{p \in primes, p>2}(1 - 1/p^2) = \frac{8}{\pi^2} \approx 81\%$. This means that even for a modest number of iterations, e.g. $\ell = 8$, we have nearly 81% success probability. These estimates are confirmed by our experiments.

Evaluation. We tried this attack on 100 different key generation efforts (2048-bit key). On average, we iterate over 81,090 steps by issuing 230,050 interrupts per attack in 800ms. The average time to collect a trace is about a second and the attack takes about 30 seconds to successfully recover the key for 81% of the keys when $lcm((p-1)(q-1)) \equiv (p-1)(q-1)/2^i$.

Revisiting Masking Protection. Earlier attacks required brute forcing over e [75]. Our algorithm works for arbitrary, even full length e. Thus increasing the size of e by choosing a bigger public exponent or masking is not sufficient to

mitigate our attack. Aldaya et al. [5] proposed masking e by computing $b = (er)^{-1} \bmod \lambda(N)$ for a random r such that $\gcd(r, \lambda(N)) = 1$. The private key then can be computed as $d = rb \bmod \lambda(N)$. In this proposal, it is even suggested that r can be hard coded. We tested our attack for a hard coded (known) choice of r and verified that key recovery works in this case. Alternatively, if r is not hard coded but we have a trace for the initial $\gcd(r, \lambda(N))$ computation using binary gcd, we can again decode it (with the knowledge of N) to recover r. With r recovered, the attack proceeds as before, i.e. from the execution trace of $b = (er)^{-1} \bmod \lambda(N)$ we recover p and q by running Algorithm 3 with er supplied as input instead of e. Since Algorithm 3 is agnostic with respect to the size of e, it will handle the full size er and recover p and q.

4.4 Breaking ECDSA Timing Protection

WolfSSL uses the subroutine `wc_ecc_mulmod_ex` (Listing 1) to compute the scalar multiplication $k \times G$ while generating the signature. This subroutine has built-in mitigations against side-channel attacks and implements an `always-add-and-double` algorithm by arithmetizing the conditional check for the `add`. As a result the scalar operations `add` at Line 15/18 and double at Line 16/19 will both be executed for all scalar bits. This prevents an adversary learning the nonce k bit by bit. The second countermeasure that is implemented in this implementation aims to protect against attacks exploiting the bit length of the nonce [16, 48]. This is done by executing a sequence of dummy operations for each leading zero bit. While these dummy operations mitigate side channels like data cache attacks, page-level attacks,

Table 2: Minimum number of signature samples for each bias class to reach 100% recovery success for the lattice-based key recovery on `wc_ecc_mulmod_ex` of ECDSA, with lattice reduction time L-TIME and trace collection time T-TIME.

LZBS	DIM	L-TIME	SIGNATURES	IRQS	T-TIME
4	75	30 sec	1,200	3.9M	13.3 sec
5	58	5 sec	1,856	6.0M	20.4 sec
6	46	3 sec	2,944	9.6M	33.7 sec
7	42	2 sec	5,376	17.5M	1 min

and timing attacks, we can use COPYCAT to distinguish the branch outcome at Line 13 and leak the bit length of nonce k.

Recovering Dummy Operations. We analyze `wc_ecc_mulmod_ex` using COPYCAT. In this analysis, we count the number of instructions that are executed between consecutive accesses to the page that holds the `ecc_projective_dbl_point` subroutine. The trace shows that for one transition of basic blocks, we can observe 49 steps when the function is processing the dummy operations. As soon as the subroutine switches to the real operations, this step count will change to 46. As a result, we can use this information to determine the number of dummy executions of the `always-add-and-double` sequence from a set of traces. Since we only need to observe the first few bits in order to recover the length of the nonce, we shortened our trace collection to observe only the first 7 bits.

Lattice Attack using the Nonce Bit Length. We generated many signature traces, recovered the nonce lengths, and filtered for signatures with short nonces [13]. We followed the approach of Howgrave-Graham and Smart [33] and Benger et al. [8] to formulate the key recovery as a lattice problem.

Evaluation. We executed this attack for 10,000 signing operations. Our attack recovered the number of leading zero bits with 100% accuracy. On average, each attack issues 3244 IRQs to count 2542 steps of the scalar multiplication operation. Table 2 shows the results for key recovery using various nonce bit lengths. Since the nonce length is recovered without noise, the lattice attack is quite efficient.

5 COPYCAT-Based Side-Channel Analysis

Now that we have empirically verified through real-world attacks that COPYCAT can recover the runtime control flow of all the branches deterministically, we analyze similar cryptographic implementations in other open-source libraries including the latest Libgcrypt 1.8.5, OpenSSL 1.1.1d, and Intel IPP Crypto [35]. OpenSSL and Intel IPP Crypto are particularly important for products using Intel SGX. Intel has an official wrapper around OpenSSL, called Intel SGX-SSL [37].

Algorithm 4 Modular inversion using a variant of BEEA.

```
 1: procedure MODINV(u, modulus v)
 2:     u₁ ← 1, u₂ ← 0, u₃ ← u
 3:     v₁ ← v, v₂ ← u₁ − u, v₃ ← v
 4:     if isOdd(u) then
 5:         t₁ ← 0, t₂ ← −1, t₃ ← −v
 6:     else
 7:         t₁ ← 1, t₂ ← 0, t₃ ← u
 8:     while t₃ ≠ 0 do
 9:         while isEven(t₃) do
10:             if isOdd(t₁) or isOdd(t₂) then
11:                 t₁ ← t₁ + v, t₂ ← t₂ − u
12:             t₁ ← t₁/2, t₂ ← t₂/2, t₃ ← t₃/2
13:         if t₃ > 0 then
14:             u₁ ← t₁, u₂ ← t₂, u₃ ← t₃
15:         else
16:             v₁ ← v − t₁, v₂ ← −u − t₂, v₃ ← −t₃
17:         t₁ ← u₁ − v₁, t₂ ← u₂ − v₂, t₃ ← u₃ − v₃
18:         if t₁ < 0 then t₁ ← t₁ + v, t₂ ← t₂ − u
    return u₁
```

The current version of Intel SGX-SSL is based on the stable OpenSSL 1.1.1d. Intel IPP Crypto is the official cryptographic library by Intel, and it is deployed in many Intel products including Intel SGX SDK [39]. Table 3 summarizes our findings in this paper regarding vulnerable code paths.

5.1 Libgcrypt Analysis

Libgcrypt uses a custom implementation of the extended Euclidean algorithm to compute modular inverses (Algorithm 4). This algorithm is based on an exercise from The Art of Computer Programming [43, Vol II, §4.5.2, Alg X]. The algorithm is an adaptation of Algorithm X to use the efficient divide by 2 reduction steps in the Binary Euclidean Algorithm. The algorithm computes a vector (u_1, u_2, u_3) such that $uu_1 + vu_2 = u_3 = \gcd(u, v)$ using auxiliary vectors $(v_1, v_2, v_3), (t_1, t_2, t_3)$. The iterations preserve the invariants $ut_1 + vt_2 = t_3$, $uu_1 + vu_2 = u_3$ and $uv_1 + vv_2 = v_3$. This algorithm is used in numerous places for secret operations.

$k^{-1} \bmod n$ **in DSA, ECDSA and ElGamal.** The DSA, ECDSA and ElGamal signature schemes all require computing $k^{-1} \bmod n$. In Libgcrypt, all of these computations are performed using Algorithm 4. We derive a single-trace attack similar to Section 4.2 that recovers all the branches of this algorithm during this computation. This trivially leaks k^{-1} for each of these algorithms in a single-trace attack. As a result, they are all vulnerable to the attack described in Section 4.2. Note that no masking countermeasure is used for DSA and ElGamal, and we discuss below how the masking countermeasure for ECDSA is insecure.

ECDSA Masking Countermeasure. We identified two vulnerabilities in how masking is applied during ECDSA signing in Libgcrypt, as shown in Listing 2, which leaves it

Table 3: An overview of applicability of COPYCAT on cryptographic libraries: WolfSSL, Libgcrypt, OpenSSL, IPP Crypto.

	Operation (Subroutine)	Implementation	Secret Branch	Exploitable	Computation → Vulnerable Callers	Single-Trace Attack
WolfSSL	Scalar Multiply (`wc_ecc_mulmod_ex`)	Montgomery Ladder w/ Branches	✔	✔	$(k \times G) \to$ `wc_ecc_sign_hash`	✘
	Greatest Common Divisor (`fp_gcd`)	Euclidean (Divisions)	✔	✘	N/A	N/A
	Modular Inverse (`fp_invmod`)	BEEA	✔	✔	$(k^{-1} \bmod n) \to$ `wc_DsaSign` $(q^{-1} \bmod p) \to$ `wc_MakeRsaKey` $(e^{-1} \bmod \Lambda(N)) \to$ `wc_MakeRsaKey`	✔ ✔ ✔
Libgcrypt	Greatest Common Divisor (`mpi_gcd`)	Euclidean (Divisions)	✔	✘	N/A	N/A
	Modular Inverse (`mpi_invm`)	Modified BEEA [43, Vol II, §4.5.2]	✔	✔	$(k^{-1} \bmod n) \to$ {dsa,elgamal}.c::sign,_gcry_ecc_ecdsa_sign $(q^{-1} \bmod p) \to$ generate_{std,fips,x931} $(e^{-1} \bmod \Lambda(N)) \to$ generate_{std,fips,x931}	✔ ✔ ✔
OpenSSL	Greatest Common Divisor (`BN_gcd`)	BEEA	✔	✔	$gcd(q-1, p-1) \to$ `RSA_X931_derive_ex`	✔
	Modular Inverse (`BN_mod_inverse_no_branch`)	BEEA w/ Branches	✘	N/A	N/A	N/A
IPP Crypto	Greatest Common Divisor (`ippsGcd_BN`)	Modified Lehmer's GCD	✔	?	$gcd(q-1, e) \to$ `cpIsCoPrime` $gcd(p-1, q-1) \to$ `isValidPriv1_rsa`	N/A N/A
	Modular Inverse (`cpModInv_BNU`)	Euclidean (Divisions)	✔	✘	N/A	N/A

vulnerable to attacks against Algorithm 4 and a single-trace attack during the computation of $k^{-1} \bmod n$. Using a randomly chosen blinding variable b, Libgcrypt computes the blinded signature as $s_b = k^{-1}(hb + bdr) \bmod n$. To compute the unblinded signature, it computes $s = s_b b^{-1} \bmod n$. The first vulnerability is that $k^{-1} \bmod n$ is not blinded, so a single-trace attack on this operation simply recovers the nonce k. This blinding should be modified to $s_b = (kb)^{-1}(h + xr) \bmod n$, and this can be unblinded by computing $s = s_b b \bmod n$.

```
1  mpi_mulm (dr, b, skey−>d, skey−>E.n);
2  mpi_mulm (dr, dr, r, skey−>E.n);      /* dr = d∗r mod n (blinded) */
3  mpi_mulm (sum, b, hash, skey−>E.n);
4  mpi_addm (sum, sum, dr, skey−>E.n);   /* sum = hash + (d∗r) mod n  (blinded) */
5  mpi_mulm (sum, bi, sum, skey−>E.n);   /* undo blinding by b^−1 */
6  mpi_invm (k_1, k, skey−>E.n);         /* k_1 = k^(−1) mod n  */
7  mpi_mulm (s, k_1, sum, skey−>E.n);    /* s = k^(−1)∗(hash+(d∗r)) mod n */
```

Listing 2: The masking protection for ECDSA leaves the $k^{-1} \bmod n$ operation vulnerable to our single-trace attack.

The second vulnerability is that since b needs to be inverted in this blinding scheme, Libgcrypt computes the $b^{-1} \bmod n$ using the same vulnerable implementation (Listing 3). Therefore, a single-trace attack can also recover the blinding value.

```
1  do {  _gcry_mpi_randomize (b, qbits, GCRY_WEAK_RANDOM);
2        mpi_mod (b, b, skey−>E.n);
3  } while (!mpi_invm (bi, b, skey−>E.n));
```

Listing 3: `_gcry_ecc_ecdsa_sign` computes the modular inverse of the blinding factor b using a vulnerable function.

RSA Input Masking. To avoid timing attacks, RSA decryption and signing in Libgcrypt use masking on the input ciphertext or message. For a random variable r and input ciphertext c, the decryption is performed on $m_b = (cr^e)^d \bmod n = c^d r \bmod n$. The message can then be unblinded by computing $m = m_b r^{-1} = c^d \bmod n$. Unfortunately, the $r^{-1} \bmod n$ is also computed using the vulnerable modular inverse function. As

a result, a single-trace attack can recover the blinding factor, rendering this countermeasure ineffective.

RSA Key Generation. Three RSA key generation subroutines in Libgcrypt: `generate_std`, `generate_fips` and `generate_x931` all use the vulnerable `mpi_invm` function to compute both $q^{-1} \bmod p$ and $e^{-1} \bmod \lambda(N)$, and are vulnerable to the attacks described in Section 4.3.

```
1  if (!BN_sub(r1, rsa−>p, BN_value_one())) goto err;   /* p−1 */
2  if (!BN_sub(r2, rsa−>q, BN_value_one())) goto err;   /* q−1 */
3  if (!BN_mul(r0, r1, r2, ctx)) goto err;              /* (p−1)(q−1) */
4  if (!BN_gcd(r3, r1, r2, ctx)) goto err;
```

Listing 4: `RSA_X931_derive_ex` uses `BN_gcd` to compute $\lambda(N)$, exposing p and q to our attack.

5.2 Analysis of OpenSSL

After many iterations and multiple attacks [27, 75], OpenSSL implemented a constant-time modular inversion function, `BN_mod_inverse_no_branch` for DSA, ECDSA, and RSA key generation. In various critical primitives, this function is also used to compute the GCD. However the legacy binary GCD function is still supported in the latest OpenSSL code base, version 1.1.1d, in the function `BN_gcd` (cf. Appendix Algorithm 5). The subroutine `RSA_X931_derive_ex`, which is responsible for generating RSA keys according to the X.931 standard, uses this function during the computation of $\lambda(N) = lcm(p-1, q-1) = (p-1)(q-1)/gcd(p-1, q-1)$, as shown in Listing 4. Thus we can apply our attack technique from Section 4.3 to recover the RSA private key from the computation of $gcd(p-1, q-1)$.

5.3 Analysis of Intel IPP Crypto

The *Intel IPP Crypto* library uses a conventional Euclidean algorithm to compute modular inverses. This algorithm per-

forms a series of division operations in a loop. While COPY-CAT can recover the precise number of division operations, this leakage does not seem to be exploitable during the RSA key generation [48, §6].

On the other hand, for computing the GCD, Intel IPP Crypto uses a modified version of Lehmer's GCD algorithm [63]. Lehmer's GCD algorithm and Intel's modified implementation are not constant time, and have secret-dependent branches [36]. This GCD implementation is only used during RSA key generation, where only a single-trace attack results in a vulnerability. Our analysis in Section 4.3 does not directly apply to this algorithm, and we leave the analysis and potential exploitability of this implementation for future work. This potential oversight in Intel's GCD implementation once more illustrates the intricacies of applying Intel's own recommended constant-time programming guidelines [41].

5.4 More Single-Trace Attack Evaluations

Attack on DSA, ECDSA and ElGamal (Libgcrypt). We replicated the attack in Section 4.2 using synthetic traces from Algorithm 4. We ran the attack on 100 different k^{-1} mod n and recovered k_{inv} and the secret key in all cases. The attack applies to ElGamal as well by computing the private key $x = r^{-1}(h - sk) \mod (p - 1)$.

Attack on RSA Key Generation (Libgcrypt, OpenSSL). We replicated synthetic traces of branches from OpenSSL's binary GCD algorithm executed on $\gcd(q - 1, p - 1)$. We applied Algorithm 2 with a modified test function modeling this algorithm, and applied a heuristic to match the appropriate number of trace steps to the bits guessed so far. We ran the attack using synthetic traces for 100 different 256-bit RSA keys. This key size is chosen to efficiently verify the correctness of our algorithm. Our attack successfully recovered every key. We similarly replicated the same attack as Section 4.3 with a test function following Algorithm 4. Similarly, we ran the attack using synthetic traces for 100 different 256-bit RSA keys and the attack was successful in all cases.

6 Limitations and Future Work

Vulnerable Code Patterns. COPYCAT interrupts a victim enclave precisely one instruction at a time and relies on a secondary page-table oracle to assign a spatial resolution to each instruction-granular observation. Our attack is thus only effective when the victim code contains a secret-dependent branch that accesses a different code or data page at the same instruction offset in both execution paths. In contrast to previous controlled-channel attacks [61, 72, 80], our notion of instruction-granular page access traces allows the *sequence* of code and data page visits in both branches to be identical.

We essentially only need a "marker" page that is accessed at a different relative instruction offset in the secret-dependent execution path. We found that in practice compilers generate code with different page accesses at different instruction offsets in both branches for a variety of reasons, including data or stack accesses, arithmetic operations, and subroutine calls. The evaluation of the previous sections clearly shows that the instruction-granular page access traces extracted by COPYCAT are strictly stronger, and can hence target more vulnerable code patterns, than the page fault sequences exploited by prior controlled-channel attacks. In order to not be vulnerable to COPYCAT, secret-dependent code paths should ideally be avoided altogether, or they should be explicitly aligned in such a way that both branches always access the exact same set of code and data pages for every instruction among both execution paths.

Automation Opportunities. The case-study attacks presented in this paper relied on careful manual inspection of the victim enclave source code and binary layout to identify vulnerable secret-dependent code patterns. We expect that dynamic analysis and symbolic execution approaches could further improve the effectiveness of our attacks, and increase confidence for defenders, by automating the discovery of vulnerable code patterns [73, 76] and possibly even the synthesis of proof-of-concept exploitation code. While the requirements for vulnerable code patterns are relatively clear-cut, as described above, we expect that it may be particularly challenging to automatically track the propagation of secrets and distinguish between non-secret and secret-dependent control flows [11].

Comparison to Branch Prediction Leakage. Table 1 identified branch prediction side channels [24, 34, 44] as an alternative attack vector to spy on enclave control flow at an instruction-level granularity with reasonable accuracy. In contrast to COPYCAT, however, microarchitectural leakage from branch predictors is inherently noisy and typically requires multiple runs of the victim enclave, ruling out this class of side channels for the noiseless single-trace attacks on key generation algorithms presented in this paper. Furthermore, in contrast to the architectural interrupt and paging interfaces exploited by COPYCAT, branch prediction side-channel leakage can be eradicated relatively straightforwardly by flushing branch history buffers when exiting the enclave, similar to the microcode updates Intel already distributed to flush branch predictors on enclave entry in response to Spectre threats [19].

In addition to being deterministic, our attack is significantly easier to scale and replicate, considering that branch predictors feature a complex design that may change from one microarchitecture to the other. BranchScope [24], for instance, relies on finding a heuristic through reverse engineering to probe a specific branch. This heuristic is dependent on *(i)* the state of other components like global and tournament predictors; and *(ii)* the exact binary layout of the victim program.

Previous attacks focus on distinguishing one or a small number of branches and we believe that replicating BranchScope to probe multiple branches across various targets (e.g., BEEA) would be challenging and may even be practically infeasible. COPYCAT, in contrast, is much easier to replicate, and we showed in Section 4 that our attack scales to probing the entire execution path in a single run.

7 Mitigation Strategies

Interrupt Detection. COPYCAT relies on the ability to single-step enclaved execution, which is within Intel SGX's threat model [40, 70]. While SGX enclaves remain explicitly interrupt-unaware by design, some research proposals [21, 60] retrofit hardware support for transactional memory to detect suspicious interrupt rates as a side-effect of an ongoing attack. However, such features are not commonly available on off-the-shelf SGX hardware, and they would not fundamentally address the attack surface as COPYCAT adversaries are likely to develop stealthier attack techniques [72, 74] that remain under the radar of heuristic defenses. Nevertheless, following a long line of microarchitectural attacks [34, 44, 47, 70, 71] abusing interrupts, our study provides strong evidence that interrupts may also amplify deterministic controlled-channel leakage and should be taken into account in the enclaved execution threat model. We advocate for architectural changes in the Intel SGX design and further research to rule out interrupt-driven attack surface [17].

Self-Paging. Recent work [50] proposes modifications to the Intel SGX architecture to rule out page-fault controlled channels by delegating paging decisions to the enclave. The proposed design modifies the processor to no longer report the faulting page base address to the untrusted OS and to not update "accessed" and "dirty" page-table attributes when in enclave mode. While these modifications would indeed thwart the deterministic spatial dimension of the COPYCAT instantiations described in this paper, we expect that adversaries may adapt by resorting to alternative side-channel oracles to construct instruction-granular page access patterns. A particularly promising avenue in this respect would be to combine COPYCAT interrupt counting with the distinct timing differences observed for unprotected page-table entries that were brought into the CPU cache during enclaved execution [72].

Static Code Balancing. We encourage future research in improved compile-time hardening techniques that may automatically rewrite conditional branches to always ensure a constant interrupt counting pattern, regardless of the executed code. The key requirement for such a defense would be to ensure that the adversary not only observes a secret-independent sequence of pages but also always counts the same number of instructions between page transitions. To achieve such a guarantee, the compiler would also have to be explicitly aware of macro fusion, as explained in Section 3.1, when balancing the observed instruction counts. We expect further challenges when dealing with secret-dependent loop bounds, as in our attacks of Section 4. To handle data accesses, control-flow balancing techniques could potentially be combined with existing solutions for data location randomization [14].

Constant-Time Implementation. The best practice for cryptographic implementations is to avoid secret-dependent branches and memory lookups. WolfSSL applied such a countermeasure to mitigate our attack on ECDSA (CVE-2019-19960). Bernstein and Yang proposed a constant-time GCD algorithm that can be used for applications like modular inversion [10]. However, constant-time implementations are not easy for generic non-cryptographic applications [64]. While there are tools and techniques to test [76], verify [6], and generate [12] constant-time code, the scalability and performance of these approaches is still far from settled.

Cryptographic Countermeasures. While it is preferable to avoid secret-dependent branches altogether, specific countermeasures can be applied to some cryptographic schemes. As we discuss in Section 5, masking the input of the modular inversion can mitigate our demonstrated attack if it is applied properly and the blinding value itself is not leaked. WolfSSL applies this solution to mitigate our attack on DSA (CVE-2019-19963). However, as we showed in Section 5 and Section 4.3, these countermeasures should be applied carefully in the presence of a powerful adversary.

Some operations have more secure alternative implementations. In particular, for the attack on q^{-1} mod p RSA-CRT key generation (CVE-2020-7960), Fermat's Little Theorem computes q^{p-2} mod p. As a result, the implementation can avoid modular inversion for this operation, and instead rely on a constant-time modular exponentiation implementation.

8 Conclusion

Our works show that deterministic controlled-channel adversaries are not restricted to observing enclave memory accesses at the level of coarse-grained 4 KiB pages, but can also precisely reconstruct intra-page control flow at a maximal, instruction-level granularity. We demonstrated the practicality and improved resolution of COPYCAT by discovering highly dangerous single-trace key extraction attacks in several real-world, side-channel hardened cryptographic libraries. In contrast to known microarchitectural leakage sources, the more fundamental threat of deterministic controlled-channel leakage cannot be dealt with by merely flushing or partitioning microarchitectural state and instead requires research into more principled solutions.

Acknowledgments

References

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *ACM CCS*, 2007.

[2] Alejandro Cabrera Aldaya and Billy Bob Brumley. When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA. *IACR TCHES*, 2020.

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *IEEE Security and Privacy (S&P)*, 2019.

[4] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-Timing Attacks on RSA Key Generation. *IACR TCHES*, 2019.

[5] Alejandro Cabrera Aldaya, Alejandro J Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA vulnerabilities of the binary extended Euclidean algorithm. *Journal of Cryptographic Engineering*, 2017.

[6] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time Implementations. In *USENIX Security*, 2016.

[7] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On Subnormal Floating Point and Abnormal Timing. In *IEEE Security and Privacy (S&P)*, 2015.

[8] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In *CHES*, 2014.

[9] Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *CHES*, 2017.

[10] Daniel J Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR TCHES*, 2019.

[11] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying Constant-Time Implementations by Abstract Interpretation. *Journal of Computer Security*, 2019.

[12] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security*, 2017.

[13] Dan Boneh and Ramarathnam Venkatesan. Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In *Advances in Cryptology*, 1996.

[14] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization. In *ACSAC*, 2019.

[15] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX WOOT*, 2017.

[16] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks are Still Practical. In *European Symposium on Research in Computer Security*, 2011.

[17] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Provably Secure Isolation for interruptible Enclaved Execution on Small Microprocessors. In *IEEE CSF*, 2020.

[18] Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro J Cabrera Sarmiento, and Santiago Sánchez-Solano. Side-channel analysis of the modular inversion step in the RSA key generation algorithm. *International Journal of Circuit Theory and Applications*, 2017.

[19] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *IEEE European Security and Privacy (Euro S&P)*, 2019.

[20] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *IEEE Security and Privacy (S&P)*, 2018.

[21] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *ACM Asia CCS*, 2017.

[22] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *IACR TCHES*, 2018.

[23] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *USENIX Security*, 2020.

[24] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[25] Fortanix. Self-Defending Key Management Service with Intel® Software Guard Extensions. https://bit.ly/2yWxuuD.

[26] Patrick Gallagher. Digital Signature Standard (DSS). *Federal Information Processing Standards Publications, volume FIPS*, pages 186–3, 2013.

[27] Cesar Pereida García and Billy Bob Brumley. Constant-Time Callees with Variable-Time Callers. In *USENIX Security*, 2017.

[28] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1), 2018.

[29] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution. In *International Symposium on Engineering Secure Software and Systems*, 2018.

[30] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *USENIX ATC*, 2017.

[31] Nadia Heninger and Hovav Shacham. Reconstructing RSA Private Keys from Random Key Bits. In *Annual International Cryptology Conference*, 2009.

[32] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018.

[33] Nick A Howgrave-Graham and Nigel P. Smart. Lattice Attacks on Digital Signature Schemes. *Designs, Codes and Cryptography*, 23(3), 2001.

[34] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. *IACR TCHES*, 2020.

[35] Intel. Intel IPP Crypto Library (commit ad2ad95). https://github.com/intel/ipp-crypto.

[36] Intel. Intel Lehmer'c GCD Implementation sources/ippcp/pcpbnarithgcd.c. https://github.com/intel/ipp-crypto/blob/b6848dc/sources/ippcp/pcpbnarithgcd.c#L54.

[37] Intel. Intel SGX SSL. https://github.com/intel/intel-sgx-ssl.

[38] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://intel.ly/2UbLwk2.

[39] Intel. Intel Software Guard Extensions Developer Guide. https://intel.ly/3dr6PFV, June 2017.

[40] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://intel.ly/2UeQjBm, May 2019.

[41] Intel. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. https://intel.ly/2WDDS3y, 2020.

[42] Deokjin Kim, Daehee Jang, Minjoon Park, Yunjong Jeong, Jonghwan Kim, Seokjin Choi, and Brent Byunghoon Kang. SGX-LEGO: Fine-grained SGX controlled-channel attack and its countermeasure. *computers & security*, 82, 2019.

[43] Donald E Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, 2014.

[44] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*, 2017.

[45] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[46] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX. In *CT-RSA*, 2018.

[47] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *CHES*, 2017.

[48] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *USENIX Security*, 2020.

[49] Phong Q Nguyen and Igor E Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Designs, codes and cryptography*, 2003.

[50] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing Controlled Channels with Self-Paging Enclaves. In *EuroSys*, 2020.

[51] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make Sure DSA Signing Exponentiations Really Are Constant-Time. In *ACM CCS*, 2016.

[52] Matthieu Rivain. Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves. *IACR Cryptology ePrint Archive*, 2011.

[53] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 1978.

[54] Keegan Ryan. Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone. In *ACM CCS*, 2019.

[55] Keegan Ryan. Return of the Hidden Number Problem. *IACR TCHES*, pages 146–168, 2019.

[56] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *NDSS*, 2018.

[57] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM CCS*, 2019.

[58] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[59] Atle Selberg. An elementary proof of the prime-number theorem. *Annals of Mathematics*, 1949.

[60] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*, 2017.

[61] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing Page Faults from Telling Your Secrets. In *ACM Asia CCS*, 2016.

[62] Signal. Private Contact Discovery Service. https://bit.ly/2MmP8uy.

[63] Jonathan Sorenson. An analysis of Lehmer's Euclidean GCD algorithm. In *Proceedings of the 1995 international symposium on Symbolic and algebraic computation*, pages 254–258, 1995.

[64] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *ACM CCS*, 2013.

[65] Raoul Strackx and Frank Piessens. The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks. *arXiv preprint arXiv:1712.08519*, 2017.

[66] Daniel Townley and Dmitry Ponomarev. SMT-COP: Defeating Side-Channel Attacks on Execution Units in SMT Processors. In *ACM International Conference on Parallel Architectures and Compilation Techniques*, 2019.

[67] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[68] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE Security and Privacy (S&P)*, 2020.

[69] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *ACM CCS*, 2019.

[70] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017.

[71] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *ACM CCS*, 2018.

[72] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*, 2017.

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

[74] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *ACM CCS*, 2017.

[75] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against RSA key generation in Intel SGX SSL. In *ACM Asia CCS*, 2018.

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

[77] WikiChip. Macro-Operation Fusion (MOP Fusion). https://en.wikichip.org/wiki/macro-operation_fusion, 2020.

[78] WolfSSL. WolfSSL Intel SGX + FIPS 140-2!

[79] Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. vCAT: Dynamic Cache Management using CAT Virtualization. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[80] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Security and Privacy (S&P)*, 2015.

[81] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, 2014.

[82] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[83] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant-time RSA. *Journal of Cryptographic Engineering*, 2017.

[84] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM CCS*, 2012.

A   Appendix

A.1   OpenSSL GCD Algorithm

Algorithm 5 shows the binary GCD algorithm in OpenSSL.

Algorithm 5 OpenSSL Binary GCD Algorithm.

1: **procedure** GCD(a, b)
2:     $s \leftarrow 0$
3:     **if** $a < b$ **then** $a, b \leftarrow b, a$
4:     **while** $b \neq 0$ **do**
5:         **if** $isOdd(a)$ **then**
6:             **if** $isOdd(b)$ **then**   ▷ a is odd, b is odd
7:                 $a \leftarrow a - b, a \leftarrow a/2$
8:                 **if** $a < b$ **then** $a, b \leftarrow b, a$
9:             **else**   ▷ a is odd, b is even
10:                 $b \leftarrow b/2$
11:                 **if** $a < b$ **then** $a, b \leftarrow b, a$
12:         **else**
13:             **if** $isOdd(b)$ **then**   ▷ a is even, b is odd
14:                 $a \leftarrow a/2$
15:                 **if** $a < b$ **then** $a, b \leftarrow b, a$
16:             **else**   ▷ a is even, b is even
17:                 $a \leftarrow a/2, b \leftarrow b/2, s \leftarrow s + 1$
18:         **if** $s > 0$ **then** $a \leftarrow a * (2^s)$
19:         **return** a

A.2   Branch Shadow-Resistant Code Attack

Listing 5 provides an elementary example function with secret-dependent branches. We provide the corresponding assembly output in Listing 6, as produced by the LLVM-based, open-source compiler mitigation pass [32] against branch shadowing attacks, described in Section 3.3. We passed the `-mllvm -x86-branch-conversion` and `-mllvm -x86-bc-dummy-instr` options to enable both rewriting of conditional branches via the trampoline area and protection against timing attacks via dummy instruction balancing. Note that the randomizer is not integrated in the open-source release, and all code blocks on the trampoline area would still have to be randomly re-shuffled at runtime to protect against branch-shadowing attacks. To achieve sufficient entropy, trampoline areas have to be larger than 4 KiB [32], and hence the trampoline will occupy at least one separate page.

We reveal control flow in the instrumented code of Listing 6 using COPYCAT as follows. In the case where the secret-dependent if condition is true, the indirect branch at line 20 will execute the single-instruction `jmp_if` block on the trampoline page, followed by 4 instructions on the instrumented code page, totaling 5 instructions before reaching the `end_if` marker. In contrast, if the if condition is false, the indirect branch at line 20 will transfer to the `skip_if` block on

```
void my_func(int a) {
    if (a != 0) block1++; else block2++;
    block3++;
}
```

Listing 5: Sample code snippet with conditional branching.

```
1          jmp     my_func /***  BEGIN TRAMPOLINE ***/
2  jmp_done:  jmp     done
3  jmp_done2: jmp     done
4  skip_else: add     $0x0,%r13b   # compensating dummy
5          lea     jmp_done2(%rip),%r15
6          jmp     end_else
7  jmp_else:  jmp     else
8  skip_if:   add     $0x0,%r13b   # compensating dummy
9          add     $0x0,%r13b   # compensating dummy
10         lea     jmp_else(%rip),%r15
11         jmp     end_if
12 jmp_if:    jmp     if      /***  END TRAMPOLINE ***/
13 my_func:   push    %rbp
14         mov     %rsp,%rbp
15         mov     %edi,-0x4(%rbp)
16         cmpl    $0x0,-0x4(%rbp)
17         lea     jmp_if(%rip),%r15
18         lea     skip_if(%rip),%r13
19         cmove   %r13,%r15
20         jmp     *%r15
21 if:       mov     block1(%rip),%eax
22         add     $0x1,%eax
23         mov     %eax,block1(%rip)
24         lea     skip_else(%rip),%r15
25 end_if:    jmp     *%r15
26 else:      mov     block2(%rip),%eax
27         add     $0x1,%eax
28         mov     %eax,block2(%rip)
29         lea     jmp_done(%rip),%r15
30 end_else:  jmp     *%r15
31 done:      mov     block3(%rip),%eax
32         add     $0x1,%eax
33         mov     %eax,block3(%rip)
34         pop     %rbp
35         ret
```

Listing 6: Hardened assembly output, corresponding to the source code in Listing 5, as produced by the open-source branch shadowing mitigation LLVM compiler pass.

the trampoline page, totaling 4 instructions before eventually reaching the `end_if` marker back on the instrumented code page. Similar unbalanced instruction counts follow for the else block.

We experimentally verified that COPYCAT adversaries can deterministically learn the if condition by merely counting instructions and observing page accesses. Moreover, because the dummy instructions do not result in exactly balanced instruction counts, as explained above, merely counting the total amount of executed instructions even suffices in itself without having to distinguish accesses to the trampoline page.