When Good Turns Evil:
Using Intel SGX to Stealthily Steal Bitcoins

Michael Schwarz, Moritz Lipp

michael.schwarz@iaik.tugraz.at, moritz.lipp@iaik.tugraz.at

Abstract

In our talk, we show that despite all doubts, it is practical to implement malware inside SGX. Moreover, this malware uses the protection features of SGX to hide itself from all state-of-the-art detection mechanisms. We show that an unprivileged user can execute malware inside an SGX enclave that uses a cache attack to extract a secret RSA key from a co-located enclave. Our malware does not use any kernel component, privileges, or operating system modifications to proxy commands from the enclave. Instead, we built novel techniques to mount the attack without operating system support. For a code reviewer, our enclave code looks like a benign series of simple memory accesses and loops.

We demonstrate that this attack is practical by mounting a cross-enclave attack to recover a full 4096-bit RSA key used in a secure signature process. This scenario can be found in real-world situations for Bitcoin wallets that are implemented inside SGX to protect the private key. With an SGX enclave, existing detection techniques (Herath and Fogh, Black-Hat USA 2015) are not applicable anymore. The main takeaway is that SGX helps attackers in hiding their malware, without even requiring any privileges (i.e., no root privileges).

Additionally, so-called double fetch bugs are problems in APIs which can often be exploited to hijack the program flow inside a higher-privileged domain, such as given by the enclave. We show that cache attacks can be used to dynamically detect such vulnerabilities in secure enclaves without access to its code or even the binary.

Furthermore, the cache can be used as a primitive to reliably exploit such vulnerabilities, allowing to leak secrets such as private keys from enclaves. In our live demonstration, we show that SGX is not a miracle cure for badly written software and high-quality software is still required to protect secret information.

1 Overview

This whitepaper does not only cover the topics of our talk but also provides more technical details. It provides more detailed insight into the attack techniques presented in the talk and also provides a more thorough evaluation of all results. Furthermore, the whitepaper contains additional attack scenarios, such as the applicability of cache attacks from SGX applications which are isolated using

Docker containers. We also describe double-fetch scenarios in different environments, such as the Linux kernel and the ARM TrustZone, and how double-fetch detection can complement state-of-the-art fuzzing approaches.

This whitepaper consists of two parts. The first part was published as a paper at the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment 2017 with the title "Malware Guard Extension: Using SGX to Conceal Cache Attacks". The paper describes how cache attacks (specifically Prime+Probe) can be mounted from SGX enclaves, to attack the outside world as well as other co-located enclaves. These scenarios are both evaluated in a native environment, as well as across Docker containers. In all scenarios, we are able to extract a 4096-bit RSA key from an enclave with only 11 traces.

The second part was published as a paper at the 13th ACM ASIA Conference on Information, Computer and Communications Security 2018 with the title "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features". The paper discusses an automated dynamic method for detecting and exploiting double-fetch bugs using the cache as a side channel. The target is mainly the Linux kernel, to detect such bugs in the syscall interfaces, by complementing syscall fuzzers. However, it also generalizes the method for a wide range of scenarios, including Intel SGX and ARM TrustZone. As presented in the talk, the double-fetch detection using a cache-based side-channel allows detecting and reliably exploiting double-fetch bugs even if neither the source code nor the binary is available for analysis. Although the technique might seem odd to some researchers, it is a practical method to detect such bugs in these cases where no other analysis method is applicable. Finally, the paper presents a novel method to prevent the exploitation of double-fetch bugs, as it is also briefly discussed in the talk. We present a library which leverages Intel TSX, an instruction set extension implementing hardware transactional memory, to group multiple data fetches into one atomic operation, entirely preventing any exploitation attempt.

The main takeaways of both the talk and the whitepaper are as follows.

1. Despite all doubts, malware can be implemented in SGX.
2. Security features can be abused by attackers to better hide their attacks.
3. Bad code is bad code, independent of the execution environment. Writing high-quality code is still necessary to make it secure.

Simply put, combining exploitable code with Intel SGX only results in exploitable SGX enclaves.

Malware Guard Extension: Using SGX to Conceal Cache Attacks

Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard

Graz University of Technology, Austria

Abstract. In modern computer systems, user processes are isolated from each other by the operating system and the hardware. Additionally, in a cloud scenario it is crucial that the hypervisor isolates tenants from other tenants that are co-located on the same physical machine. However, the hypervisor does not protect tenants against the cloud provider and thus the supplied operating system and hardware. Intel SGX provides a mechanism that addresses this scenario. It aims at protecting user-level software from attacks from other processes, the operating system, and even physical attackers.

In this paper, we demonstrate fine-grained software-based side-channel attacks from a malicious SGX enclave targeting co-located enclaves. Our attack is the first malware running on real SGX hardware, abusing SGX protection features to conceal itself. Furthermore, we demonstrate our attack both in a native environment and across multiple Docker containers. We perform a *Prime+Probe* cache side-channel attack on a co-located SGX enclave running an up-to-date RSA implementation that uses a constant-time multiplication primitive. The attack works although in SGX enclaves there are no timers, no large pages, no physical addresses, and no shared memory. In a semi-synchronous attack, we extract 96 % of an RSA private key from a single trace. We extract the full RSA private key in an automated attack from 11 traces.

1 Introduction

Modern operating systems isolate user processes from each other to protect secrets in different processes. Such secrets include passwords stored in password managers or private keys to access company networks. Leakage of these secrets can compromise both private and corporate systems. Similar problems arise in the cloud. Therefore, cloud providers use virtualization as an additional protection using a hypervisor. The hypervisor isolates different tenants that are co-located on the same physical machine. However, the hypervisor does not protect tenants against a possibly malicious cloud provider.

Although hypervisors provide functional isolation, side-channel attacks are often not considered. Consequently, researchers have demonstrated various side-channel attacks, especially those exploiting the cache [15]. Cache side-channel attacks can recover cryptographic secrets, such as AES [29] and RSA [33] keys, across virtual machine boundaries.

Intel introduced a new hardware extension SGX (Software Guard Extensions) [27] in their CPUs, starting with the Skylake microarchitecture. SGX is an isolation mechanism, aiming at protecting code and data from modification or disclosure even if all privileged software is malicious [10]. This protection uses special execution environments, so-called enclaves, which work on memory areas that are isolated from the operating system by the hardware. The memory area used by the enclaves is encrypted to protect application secrets from hardware attackers. Typical use cases include password input, password managers, and cryptographic operations. Intel recommends storing cryptographic keys inside enclaves and claims that side-channel attacks "are thwarted since the memory is protected by hardware encryption" [25].

Hardware-supported isolation also led to fear of super malware inside enclaves. Rutkowska [44] outlined a scenario where an enclave fetches encrypted malware from an external server and executes it within the enlave. In this scenario, it is impossible to debug, reverse engineer, or analyze the executed malware in any way. Costan et al. [10] eliminated this fear by arguing that enclaves always run with user space privileges and can neither issue syscalls nor perform any I/O operations. Moreover, SGX is a highly restrictive environment for implementing cache side-channel attacks. Both state-of-the-art malware and side-channel attacks rely on several primitives that are not available in SGX enclaves.

In this paper, we show that it is very well possible for enclave malware to attack its hosting system. We demonstrate a cross-enclave cache attack from within a malicious enclave that is extracting secret keys from co-located enclaves. Our proof-of-concept malware is able to recover RSA keys by monitoring cache access patterns of an RSA signature process in a semi-synchronous attack. The malware code is completely invisible to the operating system and cannot be analyzed due to the isolation provided by SGX. We present novel approaches to recover physical address bits, as well as to recover high-resolution timing in absence of the timestamp counter, which has an even higher resolution than the native one. In an even stronger attack scenario, we show that an additional isolation using Docker containers does not protect against this kind of attack.

We make the following contributions:

1. We demonstrate that, despite the restrictions of SGX, cache attacks can be performed from within an enclave to attack a co-located enclave.
2. By combining DRAM and cache side channels, we present a novel approach to recover physical address bits even if 2 MB pages are unavailable.
3. We obtain high-resolution timestamps in enclaves without access to the native timestamp counter, with an even higher resolution than the native one.
4. In an automated end-to-end attack on the wide-spread *mbedTLS* RSA implementation, we extract 96 % of an RSA private key from a single trace.

Section 2 presents the required background. Section 3 outlines the threat model and attack scenario. Section 4 describes the measurement methods and the online phase of the malware. Section 5 explains the offline-phase key recovery. Section 6 evaluates the attack against an up-to-date RSA implementation. Section 7 discusses several countermeasures. Section 8 concludes our work.

2 Background

2.1 Intel SGX in Native and Virtualized Environments

Intel Software Guard Extensions (SGX) are a new set of x86 instructions introduced with the Skylake microarchitecture. SGX allows protecting the execution of user programs in so-called enclaves. Only the enclave can access its own memory region, any other access to it is blocked by the CPU. As SGX enforces this policy in hardware, enclaves do not need to rely on the security of the operating system. In fact, with SGX the operating system is generally not trusted. By doing sensitive computation inside an enclave, one can effectively protect against traditional malware, even if such malware has obtained kernel privileges. Furthermore, it allows running secret code in a cloud environment without trusting hardware and operating system of the cloud provider.

An enclave resides in the virtual memory area of an ordinary application process. This virtual memory region of the enclave can only be backed by physically protected pages from the so-called Enclave Page Cache (EPC). The EPC itself is a contiguous physical block of memory in DRAM that is encrypted transparently to protect against hardware attacks.

Loading of enclaves is done by the operating system. To protect the integrity of enclave code, the loading procedure is measured by the CPU. If the resulting measurement does not match the value specified by the enclave developer, the CPU will refuse to run the enclave.

Since enclave code is known to the (untrusted) operating system, it cannot carry hard-coded secrets. Before giving secrets to an enclave, a provisioning party has to ensure that the enclave has not been tampered with. SGX therefore provides remote attestation, which proves correct enclave loading via the aforementioned enclave measurement.

At the time of writing, no hypervisor with SGX support was available. However, Arnautov et al. [4] proposed to combine Docker containers with SGX to create secure containers. Docker is an operating-system-level virtualization software that allows applications to run in separate containers. It is a standard runtime for containers on Linux which is supported by multiple public cloud providers. Unlike virtual machines, Docker containers share the kernel and other resources with the host system, requiring fewer resources than a virtual machine.

2.2 Microarchitectural Attacks

Microarchitectural attacks exploit hardware properties that allow inferring information on other processes running on the same system. In particular, cache attacks exploit the timing difference between the CPU cache and the main memory. They have been the most studied microarchitectural attacks for the past 20 years, and were found to be powerful to derive cryptographic secrets [15]. Modern attacks target the last-level cache, which is shared among all CPU cores. Last-level caches (LLC) are usually built as n-way set-associative caches. They consist of S cache sets and each cache set consists of n cache ways with a size of

64 B. The lowest 6 physical address bits determine the byte offset within a cache way, the following $\log_2 S$ bits starting with bit 6 determine the cache set.

Prime+Probe is a cache attack technique that has first been used by Osvik et al. [39]. In a *Prime+Probe* attack, the attacker constantly primes (*i.e.*, evicts) a cache set and measures how long this step took. The runtime of the prime step is correlated to the number of cache ways that have been replaced by other programs. This allows deriving whether or not a victim application performed a specific secret-dependent memory access. Recent work has shown that this technique can even be used across virtual machine boundaries [33, 35].

To prime (*i.e.*, evict) a cache set, the attacker uses n addresses in same cache set (*i.e.*, an *eviction set*), where n depends on the cache replacement policy and the number of ways. To minimize the amount of time the prime step takes, it is necessary to find a minimal n combined with a fast access pattern (*i.e.*, an *eviction strategy*). Gruss et al. [18] experimentally found efficient eviction strategies with high eviction rates and a small number of addresses. We use their eviction strategy on our Skylake test machine throughout the paper.

Pessl et al. [42] found a similar attack through DRAM modules. Each DRAM module has a row buffer that holds the most recently accessed DRAM row. While accesses to this buffer are fast, accesses to other memory locations in DRAM are much slower. This timing difference can be exploited to obtain fine-grained information across virtual machine boundaries.

2.3 Side-Channel Attacks on SGX

Intel claims that SGX features impair side-channel attacks and recommends using SGX enclaves to protect password managers and cryptographic keys against side channels [25]. However, there have been speculations that SGX could be vulnerable to side-channel attacks [10]. Xu et al. [50] showed that SGX is vulnerable to page fault side-channel attacks from a malicious operating system [1].

SGX enclaves generally do not share memory with other enclaves, the operating system or other processes. Thus, any attack requiring shared memory is not possible, e.g., *Flush+Reload* [51]. Also, DRAM-based attacks cannot be performed from a malicious operating system, as the hardware prevents any operating system accesses to DRAM rows in the EPC. However, enclaves can mount DRAM-based attacks on other enclaves because all enclaves are located in the same physical EPC.

In concurrent work, Brasser et al. [8], Moghimi et al. [37] and Götzfried et al. [17] demonstrated cache attacks on SGX relying on a malicious operating system.

2.4 Side-Channel Attacks on RSA

RSA is widely used to create asymmetric signatures, and is implemented by virtually every TLS library, such as OpenSSL or *mbedTLS*, which is used for instance in cURL and OpenVPN. RSA essentially involves modular exponentiation with a private key, typically using a square-and-multiply algorithm. An unprotected implementation of square-and-multiply is vulnerable to a variety of side-channel attacks, in which an attacker learns the exponent by distinguishing the square step from the multiplication step [15, 51]. *mbedTLS* uses a windowed square-and-multiply routine for the exponentiation. Liu et al. [33] showed that if an attack on a window size of 1 is possible, the attack can be extended to arbitrary window sizes.

Earlier versions of *mbedTLS* were vulnerable to a timing side-channel attack on RSA-CRT [3]. Due to this attack, current versions of *mbedTLS* implement a constant-time Montgomery multiplication for RSA. Additionally, instead of using a dedicated square routine, the square operation is carried out using the multiplication routine. Thus, there is no leakage from a different square and multiplication routine as exploited in previous attacks on square-and-multiply algorithms [33, 51]. However, Liu et al. [33] showed that the secret-dependent accesses to the buffer b still leak the exponent. Boneh et al. [7] and Blömer et al. [6] recovered the full RSA private key if only parts of the key bits are known.

3 Threat Model and Attack Setup

In this section, we present our threat model. We demonstrate a malware that circumvents SGX and Docker isolation guarantees. We successfully mount a *Prime+Probe* attack on an RSA signature computation running inside a different enclave, on the outside world, and across container boundaries.

3.1 High-Level View of the Attack

In our threat model, both the attacker and the victim are running on the same physical machine. The machine can either be a user's local computer or a host in the cloud. In the cloud scenario, the victim has its enclave running in a Docker container to provide services to other applications running on the host. Docker containers are well supported on many cloud providers, e.g., Amazon [13] or Microsoft Azure [36]. As these containers are more lightweight than virtual machines, a host can run up to several hundred containers simultaneously. Thus, the attacker has good chances to get a co-located container on a cloud provider.

Figure 1 gives an overview of our native setup. The victim runs a cryptographic computation inside the enclave to protect it against any attacks. The attacker tries to stealthily extract secrets from this victim enclave. Both the attacker and the victim use Intel SGX features and thus are subdivided into two parts, the enclave and loader, *i.e.*, the main program instantiating the enclave.

The attack is a multi-step process that can be divided into an online and an offline phase. Section 4 describes the online phase, in which the attacker first locates the victim's cache sets that contain the secret-dependent data of the RSA private key. The attacker then monitors the identified cache sets while triggering a signature computation. Section 5 gives a detailed explanation of the offline phase in which the attacker recovers a private key from collected traces.

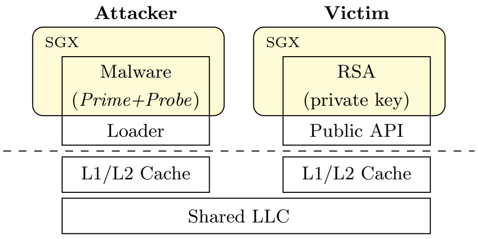


Fig. 1: The threat model: both attacker and victim run on the same physical machine in different SGX enclaves.

3.2 Victim

The victim is an unprivileged program that uses SGX to protect an RSA signing application from both software and hardware attackers. Both the RSA implementation and the private key reside inside the enclave, as suggested by Intel [25]. Thus, they can never be accessed by system software or malware on the same host. Moreover, memory encryption prevents physical information leakage in DRAM. The victim uses the RSA implementation of the widely deployed *mbedTLS* library. The *mbedTLS* library implements a windowed square-and-multiply algorithm, that relies on constant-time Montgomery multiplications. The window size is fixed to 1, as suggested by the official knowledge base [2]. The victim application provides an API to compute a signature for provided data.

3.3 Attacker

The attacker runs an unprivileged program on the same host machine as the victim. The goal of the attacker is to stealthily extract the private key from the victim enclave. Therefore, the attacker uses the API provided by the victim to trigger signature computations.

The attacker targets the exponentiation step of the RSA implementation. The attack works on arbitrary window sizes [33], including window size 1. To prevent information leakage from function calls, *mbedTLS* uses the same function (`mpi_montmul`) for both the square and the multiply operation. The `mpi_montmul` takes two parameters that are multiplied together. For the square operation, the function is called with the current buffer as both arguments. For the multiply operation, the current buffer is multiplied with a buffer holding the multiplier. This buffer is allocated in the calling function `mbedtls_mpi_exp_mod` using `calloc`. Due to the deterministic behavior of the tlibc `calloc` implementation, the used buffers always have the same virtual and physical addresses and thus the same cache sets. The attacker can therefore mount a *Prime+Probe* attack on the cache sets containing the buffer.

In order to remain stealthy, all parts of the malware that contain attack code reside inside an SGX enclave. The enclave can protect the encrypted real attack code by only decrypting it after a successful remote attestation after which the enclave receives the decryption key. As pages in SGX can be mapped as writable and executable, self-modifying code is possible and therefore code can be encrypted. Consequently, the attack is completely stealthy and invisible from anti-virus software and even from monitoring software running in ring 0. Note that our proof-of-concept implementation does not encrypt the attack code as this has no impact on the attack.

The loader does not contain any suspicious code or data, it is only required to start the enclave and send the exfiltrated data to the attacker.

3.4 Operating System and Hardware

Previous work was mostly focused on attacks on enclaves from untrusted cloud operating systems [10, 46]. However, in our attack we do not make any assumptions on the underlying operating system, *i.e.*, we do not rely on a malicious operating system. Both the attacker and the victim are unprivileged user space applications. Our attack works on a fully-patched recent operating system with no known software vulnerabilities, *i.e.*, the attacker cannot elevate privileges.

We expect the cloud provider to run state-of-the-art malware detection software. We assume that the malware detection software is able to monitor the behavior of containers and inspect the content of containers. Moreover, the user can run anti-virus software and monitor programs inside the container. We assume that the protection mechanisms are either signature-based, behavioral-based, heuristics-based or use performance counters [12, 21].

Our only assumption on the hardware is that attacker and victim run on the same host system. This is the case on both personal computers and on co-located Docker instances in the cloud. As SGX is currently only available on Intel Skylake CPUs, it is valid to assume that the host is a Skylake system. Consequently, we know that the last-level cache is shared between all CPU cores.

4 Extracting Private Key Information

In this section, we describe the online phase of our attack. We first build primitives necessary to mount this attack. Then we show in two steps how to locate and monitor cache sets to extract private key information.

4.1 Attack Primitives in SGX

Successful *Prime+Probe* attacks require two primitives: a high-resolution timer to distinguish cache hits and misses and a method to generate an eviction set for arbitrary cache sets. Due to the restrictions of SGX enclaves, implementing *Prime+Probe* in enclaves is not straight-forward. Therefore, we require new techniques to build a malware from within an enclave.

High-resolution Timer. The unprivileged `rdtsc` and `rdtscp` instructions, which read the timestamp counter, are usually used for fine-grained timing outside enclaves. In SGX, these instructions are not permitted inside an enclave, as they might cause a VM exit [24]. Thus, we have to rely on a different timing source with a resolution in the order of 10 cycles to reliably distinguish cache hits from misses as well as DRAM row hits from row conflicts.

To achieve the highest number of increments, we handcraft a counter thread [31,49] in inline assembly. The counter variable has to be accessible across threads, thus it is necessary to store the counter variable in memory. Memory addresses as operands incur an additional cost of approximately 4 cycles due to L1 cache access times [23]. On our test machine, a simple counting thread executing `1: incl (%rcx); jmp 1b` achieves one increment every 4.7 cycles, which is an improvement of approximately 2 % over the best code generated by `gcc`.

We can improve the performance—and thus the resolution—further, by exploiting the fact that only the counting thread modifies the counter variable. We can omit reading the counter variable from memory. Therefore, we introduce a "shadow counter variable" which is always held in a CPU register. The arithmetic operation (either `add` or `inc`) is performed on this register, unleashing the low latency and throughput of these instructions. As registers cannot be shared across threads, the shadow counter has to be moved to memory using the `mov` instruction after each increment. Similar to the `inc` and `add` instruction, the `mov` instruction has a latency of 1 cycle and a throughput of 0.5 cycles/instruction when copying a register to memory. The improved counting thread, `1: inc %rax; mov %rax, (%rcx), jmp 1b`, is significantly faster and increments the variable by one every 0.87 cycles, which is an improvement of 440 % over the simple counting thread. In fact, this version is even 15 % faster than the native timestamp counter, thus giving us a reliable timing source with even higher resolution. This new method might open new possibilities of side-channel attacks that leak information through timing on a sub-`rdtsc` level.

Eviction Set Generation. *Prime+Probe* relies on eviction sets, *i.e.*, we need to find virtual addresses that map to the same physical cache set. An unprivileged process cannot translate virtual to physical addresses and therefore cannot simply search for virtual addresses that fall into the same cache set. Liu et al. [33] and Maurice et al. [35] demonstrated algorithms to build eviction sets using large pages by exploiting the fact that the virtual address and the physical address have the same lowest 21 bits. As SGX does not support large pages, this approach is inapplicable. Oren et al. [38] and Gruss et al. [18] demonstrated automated methods to generate eviction sets for a given virtual address. Due to microarchitectural changes their approaches are either not applicable at all to the Skylake architecture or consume several hours on average before even starting the actual *Prime+Probe* attack.

We propose a new method to recover the cache set from a virtual address without relying on large pages. The idea is to exploit contiguous page allocation [28] and DRAM timing differences to recover DRAM row boundaries. The DRAM mapping functions [42] allow to recover physical address bits.

The DRAM organization into banks and rows causes timing differences. Alternately accessing pairs of two virtual addresses that map to the same DRAM bank but a different row is significantly slower than any other combination of

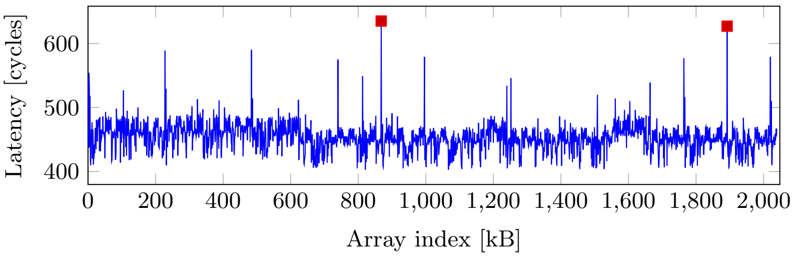


Fig. 2: Access times when alternately accessing two addresses which are 64 B apart. The (marked) high access times indicate row conflicts.

virtual addresses. Figure 2 shows the average access time for address pairs when iterating over a 2 MB array. The highest two peaks show row conflicts, *i.e.*, the row index changes while the bank, rank, and channel stay the same.

To recover physical address bits we use the reverse-engineered DRAM mapping function as shown in Table 1. Our test machine is an Intel Core i5-6200U with 12 GB main memory. The row index is determined by physical address bits 18 and upwards. Hence, the first address of a DRAM row has the least-significant 18 bits of the physical address set to '0'. To detect row borders, we scan memory sequentially for an address pair in physical proximity that causes a *row conflict*. As SGX enclave memory is allocated contiguously we can perform this scan on virtual addresses.

A virtual address pair that causes row conflicts at the beginning of a row satisfies the following constraints:

1. The least-significant 18 physical address bits of one virtual address are zero. This constitutes a DRAM row border.
2. The bank address (BA), bank group (BG), rank, and channel determine the DRAM bank and must be the same for both virtual addresses.
3. The row index must be different for both addresses to cause a row conflict.
4. The difference of the two virtual addresses has to be at least 64 B (the size of one cache line) but should not exceed 4 kB (the size of one page).

Physical address bits 6 to 17 determine the cache set which we want to recover. Hence, we search for address pairs where physical address bits 6 to 17 have the same known but arbitrary value.

		Address Bit																
		22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06
2 DIMMs	Channel				⊕	⊕					⊕	⊕			⊕	⊕		
	BG0									⊕							⊕	
	BG1	⊕				⊕												
	BA0				⊕				⊕									
	BA1		⊕				⊕											
	Rank			⊕				⊕										

Table 1: Reverse-engineered DRAM mapping functions from Pessl et al. [42].

To find address pairs fulfilling the aforementioned constraints, we modeled the mapping function and the constraints as an SMT problem and used the Z3 theorem prover [11] to provide models satisfying the constraints. The model we found yields pairs of physical addresses where the upper address is 64 B apart from the lower one. There are four such address pairs within every 4 MB block of physical memory such that each pair maps to the same bank but a different row. The least-significant bits of the physical address pairs are either (`0x3fffc0`, `0x400000`), (`0x7fffc0`, `0x800000`), (`0xbfffc0`, `0xc00000`) or (`0xffffc0`, `0x1000000`) for the lower and higher address respectively. Thus, at least 22 bits of the higher addresses least-significant bits are 0. As the cache set is determined by the bits 6 to 17, the higher address has the cache set index 0. We observe that satisfying address pairs are always 256 KB apart. Since we have contiguous memory [28], we can generate addresses mapping to the same cache set by adding multiples of 256 KB to the higher address.

In modern CPUs, the last-level cache is split into cache slices. Addresses with the same cache set index map to different cache slices based on the remaining address bits. To generate an eviction set, it is necessary to only use addresses that map to the same cache set in the same cache slice. However, to calculate the cache slice, all bits of the physical address are required [34].

As we are not able to directly calculate the cache slice, we use another approach. We add our calculated addresses from the correct cache set to our eviction set until the eviction rate is sufficiently high. Then, we try to remove single addresses from the eviction set as long as the eviction rate does not drop. Thus, we remove all addresses that do not contribute to the eviction, and the result is a minimal eviction set. Our approach takes on average 2 seconds per cache set, as we already know that our addresses map to the correct cache set. This is nearly three orders of magnitude faster than the approach of Gruss et al. [18]. Older techniques that have been comparably fast do not work on current hardware anymore due to microarchitectural changes [33, 38].

4.2 Identifying and Monitoring Vulnerable Sets

With the reliable high-resolution timer and a method to generate eviction sets, we can mount the first stage of the attack and identify the vulnerable cache sets. As we do not have any information about the physical addresses of the victim, we have to scan the last-level cache for characteristic patterns corresponding to the signature process. We consecutively mount a *Prime+Probe* attack on every cache set while the victim is executing the exponentiation step.

We can then identify multiple cache sets showing the distinctive pattern of the signature operation. The number of cache sets depends on the RSA key size. Cache sets at the buffer boundaries might be used by neighboring buffers and are more likely to be prefetched [20, 51] and thus, prone to measurement errors. Consequently, we use cache sets neither at the start nor the end of the buffer.

The measurement method is the same as for detecting the vulnerable cache sets, *i.e.*, we again use *Prime+Probe*. Due to the deterministic behavior of the

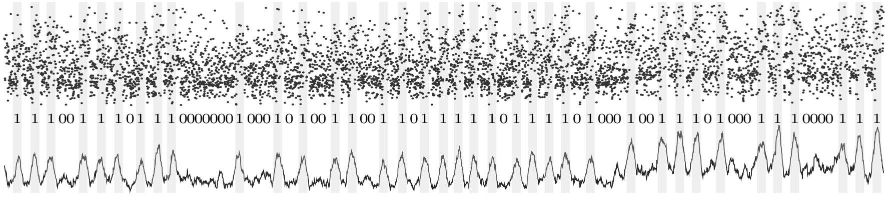


Fig. 3: A raw measurement trace over 4 000 000 cycles. The peaks in the pre-processed trace on the bottom clearly indicate ‘1’s.

heap allocation, the address of the attacked buffer does not change on consecutive exponentiations. Thus, we can collect multiple traces of the signature process.

To maintain a high sampling rate, we keep the post-processing during the measurements to a minimum. Moreover, it is important to keep the memory activity at a minimum to not introduce additional noise on the cache. Thus, we only save the timestamps of the cache misses for further post-processing. As a cache miss takes longer than a cache hit, the effective sampling rate varies depending on the number of cache misses. We have to consider this effect in the post-processing as it induces a non-constant sampling interval.

5 Recovering the Private Key

In this section, we describe the offline phase of our attack: recovering the private key from the recorded traces of the victim enclave. This can either be done inside the malware enclave or on the attacker’s server.

Ideally, an attacker would combine multiple traces by aligning them and averaging out noise. From the averaged trace, the private key can be extracted more easily. However, most noise sources, such as context switches, system activity and varying CPU clock, alter the timing, thus making trace alignment difficult. We pre-process all traces individually and extract a partial key out of each trace. These partial keys likely suffer from random insertion and deletion errors as well as from bit flips. To eliminate the errors, we combine multiple partial keys in the key recovery phase. This approach has much lower computational overhead than trace alignment since key recovery is performed on partial 4096-bit keys instead of full traces containing several thousand measurements.

Key recovery comes in three steps. First, traces are pre-processed. Second, a partial key is extracted from each trace. Third, the partial keys are merged to recover the private key. In the pre-processing step we filter and resample raw measurement data. Figure 3 shows a trace segment before (top) and after pre-processing (bottom). The pre-processed trace shows high peaks at locations of cache misses, indicating a ‘1’ in the RSA exponent.

To automatically extract a partial key from a pre-processed trace, we first run a peak detection algorithm. We delete duplicate peaks, e.g., peaks where the corresponding RSA multiplications would overlap in time. We also delete peaks that are below a certain adaptive threshold, as they do not correspond to actual multiplications. Using an adaptive threshold is necessary since neither the CPU frequency nor our timing source (the counting thread) is perfectly stable. The varying peak height is shown in the right third of Figure 3. The adaptive threshold is the median over the 10 previously detected peaks. If a peak drops below 90 % of this threshold, it is discarded. The remaining peaks correspond to the '1's in the RSA exponent and are highlighted in Figure 3. '0's can only be observed indirectly in our trace as square operations do not trigger cache activity on the monitored sets. '0's appear as time gaps in the sequence of '1' peaks, thus revealing all partial key bits. Note that since '0's correspond to just one multiplication, they are roughly twice as fast as '1's.

When a correct peak is falsely discarded, the corresponding '1' is interpreted as two '0's. Likewise, if noise is falsely interpreted as a '1', this cancels out two '0's. If either the attacker or the victim is not scheduled, we have a gap in the collected trace. However, if both the attacker and the victim are descheduled, this gap does not show up prominently in the trace since the counting thread is also suspended by the interrupt. This is an advantage of a counting thread over the use of the native timestamp counter.

In the final key recovery, we merge multiple partial keys to obtain the full key. We quantify partial key errors using the edit distance. The edit distance between a partial key and the correct key gives the number of bit insertions, deletions and flips necessary to transform the partial key into the correct key.

The full key is recovered bitwise, starting from the most-significant bit. The correct key bit is the result of the majority vote over the corresponding bit in all partial keys. To correct the current bit of a wrong partial key, we compute the edit distance to all partial keys that won the majority vote. To reduce the performance overhead, we do not calculate the edit distance over the whole partial keys but only over a lookahead window of a few bits. The output of the edit distance algorithm is a list of actions necessary to transform one key into the other. We apply these actions via majority vote until the key bit of the wrong partial key matches the recovered key bit again.

6 Evaluation

In this section, we evaluate the presented methods by building a malware enclave attacking a co-located enclave that acts as the victim. As discussed in Section 3.2, we use *mbedTLS*, in version 2.3.0.

For the evaluation, we attack a 4096-bit RSA key. The runtime of the multiplication function increases exponentially with the size of the key. Hence, larger keys improve the measurement resolution of the attacker. In terms of cache side-channel attacks, large RSA keys do not provide higher security but degrade side-channel resistance [41, 48, 51].

6.1 Native Environment

We use a Lenovo ThinkPad T460s with an Intel Core i5-6200U (2 cores, 12 cache ways) running Ubuntu 16.10 and the Intel SGX driver. Both the attacker

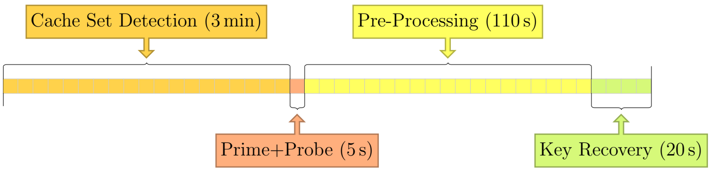


Fig. 4: A high-level overview of the average times for each step of the attack.

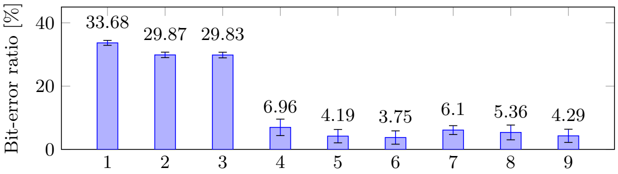


Fig. 5: The 9 cache sets that are used by a 4096-bit key and their error ratio when recovering the key from a single trace.

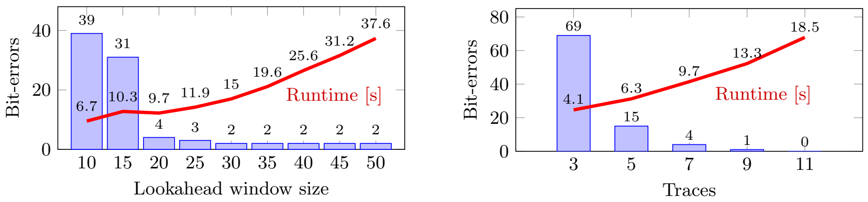
enclave and the victim enclave are running on the same machine. We trigger the signature process using the public API of the victim.

Figure 4 gives an overview of how long the individual steps of an average attack take. The runtime of automatic cache set detection varies depending on which cache sets are used by the victim. The attacked buffer spans 9 cache sets, out of which 6 show a low bit-error ratio, as shown in Figure 5. For the attack we select one of the 6 sets, as the other 3 suffer from too much noise. The noise is mainly due to the buffer not being aligned to the cache set. Furthermore, as already known from previous attacks, the hardware prefetcher can induce a significant amount of noise [20, 51].

Detecting one vulnerable cache set within all 2048 cache sets requires about 340 trials on average. With a monitoring time of 0.21 s per cache set, we require a maximum of 72 s to eventually capture a trace from a vulnerable cache set. Thus, based on our experiments, we estimate that cache set detection—if successful—always takes less than 3 min.

One trace spans 220.47 million CPU cycles on average. Typically, ‘0’ and ‘1’ bits are uniformly distributed in the key. The estimated number of multiplications is therefore half the bit size of the key. Thus, the average multiplication takes 107 662 cycles. As the *Prime+Probe* measurement takes on average 734 cycles, we do not have to slow down the victim additionally.

When looking at a single trace, we can already recover about 96 % of the RSA private key, as shown in Figure 5. For a full key recovery we combine multiple traces using our key recovery algorithm, as explained in Section 5. We first determine a reasonable lookahead window size. Figure 6a shows the performance of our key recovery algorithm for varying lookahead window sizes on 7 traces. For lookahead windows smaller than 20, bit errors are pretty high. In that case, the lookahead window is too small to account for all insertion and deletion errors,



(a) Increasing the lookahead reduces bit errors and increases runtime.

(b) Increasing the number of traces reduces bit errors and increases runtime.

Fig. 6: Relation between number of traces, lookahead window size, number of bit errors, and runtime.

causing relative shifts between the partial keys. The key recovery algorithm is unable to align partial keys correctly and incurs many wrong "correction" steps, increasing the overall runtime as compared to a window size of 20. While a lookahead window size of 20 already shows a good performance, a window size of 30 or more does not significantly reduce the bit errors. Therefore, we fixed the lookahead window size to 20.

To remove the remaining bit errors and get full key recovery, we have to combine more traces. Figure 6b shows how the number of traces affects the key recovery performance. We can recover the full RSA private key without any bit errors by combining only 11 traces within just 18.5 s. This results in a total runtime of less than 130 s for the offline key recovery process.

Generalization. Based on our experiments we deduced that attacks are also possible in a weaker scenario, where only the attacker is inside the enclave. On most computers, applications handling cryptographic keys are not protected by SGX enclaves. From the attacker's perspective, attacking such an unprotected application does not differ from attacking an enclave. We only rely on the last-level cache, which is shared among all applications, whether they run inside an enclave or not. We empirically verified that such attacks on the outside world are possible and were again able to recover RSA private keys.

Table 2 summarizes our results. In contrast to concurrent work on cache attacks on SGX [8, 17, 37], our attack is the only one that can be mounted from unprivileged user space, and cannot be detected as it runs within an enclave.

Attack on Attack from	Benign Userspace	Benign Kernel	Benign SGX Enclave
Malicious Userspace	✓ [33, 39]	✓ [22]	✓ **new**
Malicious Kernel	—	—	✓ **new** [8, 17, 37]
Malicious SGX Enclave	✓ **new**	✓ **new**	✓ **new**

Table 2: Our results show that cache attacks can be mounted successfully in the shown scenarios.

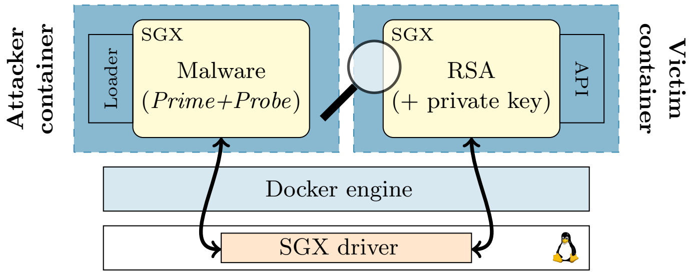


Fig. 7: Running the SGX enclaves inside Docker containers to provide further isolation. The host provides both containers access to the same SGX driver.

6.2 Virtualized Environment

We now show that the attack also works in a virtualized environment. As described in Section 2.1, no hypervisor with SGX support was available at the time of our experiments. Instead of full virtualization using a virtual machine, we used lightweight Docker containers, as used by large cloud providers, e.g., Amazon [13] or Microsoft Azure [36]. To enable SGX within a container, the host operating system has to provide SGX support. The SGX driver is then simply shared among all containers. Figure 7 shows our setup where the SGX enclaves communicate directly with the SGX driver of the host operating system. Applications running inside the container do not experience any difference to running on a native system.

Considering the performance within Docker, only I/O operations and network access have a measurable overhead [14]. Operations that only depend on memory and CPU do not see any performance penalty, as these operations are not virtualized. Thus, caches are also not affected by the container.

We were successfully able to attack a victim from within a Docker container without any changes in the malware. We can even perform a cross-container attack, *i.e.*, both the malware and the victim are running inside different containers, without any changes. As expected, we require the same number of traces for a full key recovery. Hence, containers do not provide additional protection against our malware at all.

7 Countermeasures

Most existing countermeasures cannot be applied to a scenario where a malicious enclave performs a cache attack and no assumptions about the operating system are made. In this section, we discuss 3 categories of countermeasures, based on where they ought to be implemented.

7.1 Source Level

A generic side-channel protection for cryptographic operations (e.g., RSA) is exponent blinding [30]. It will prevent the proposed attack, but other parts of the signature process might still be vulnerable to an attack [45]. More generally bit slicing can be applied to a wider range of algorithms to protect against timing side channels [5, 47]

7.2 Operating System Level

Implementing countermeasures against malicious enclave attacks on the operating system level requires trusting the operating system. This would weaken the trust model of SGX enclaves significantly, but in some threat models this can be a viable solution. However, we want to discuss the different possibilities, in order to provide valuable information for the design process of future enclave systems.

Detecting Malware. One of the core ideas of SGX is to remove the cloud provider from the root of trust. If the enclave is encrypted and only decrypted after successful remote attestation, the cloud provider has no way to access the secret code inside the enclave. Also, heuristic methods, such as behavior-based detection, are not applicable, as the malicious enclave does not rely on malicious API calls or user interaction which could be monitored. However, eliminating this core feature of SGX could mitigate malicious enclaves in practice, as the enclave binary or source code could be read by the cloud provider and scanned for malicious activities.

Herath and Fogh [21] proposed to use hardware performance counters to detect cache attacks. Subsequently, several other approaches instrumenting performance counters to detect cache attacks have been proposed [9, 19, 40]. However, according to Intel, SGX enclave activity is not visible in the thread-specific performance counters [26]. We verified that even performance counters for last-level cache accesses are disabled for enclaves. The performance counter values are three orders of magnitude below the values as compared to native code. There are no cache hits and misses visible to the operating system or any application (including the host application). This makes it impossible for current anti-virus software and other detection mechanisms to detect malware inside the enclave.

Enclave Coloring. We propose enclave coloring as an effective countermeasure against cross-enclave attacks. Enclave coloring is a software approach to partition the cache into multiple smaller domains. Each domain spans over multiple cache sets, and no cache set is included in more than one domain. An enclave gets one or more cache domains assigned exclusively. The assignment of domains is either done by the hardware or by the operating system. Trusting the operating system contradicts one of the core ideas of SGX [10]. However, if the operating system is trusted, this is an effective countermeasure against cross-enclave cache attacks.

If implemented in software, the operating system can split the last-level cache through memory allocation. The cache set index is determined by physical address bits below bit 12 (the page offset) and bits > 12 which are not visible to the enclave application and can thus be controlled by the operating system. We call these upper bits a color. Whenever an enclave requests pages from the operating system (we consider the SGX driver as part of the operating system), it will only get pages with a color that is not present in any other enclave. This coloring ensures that two enclaves cannot have data in the same cache set, and therefore a *Prime+Probe* attack is not possible across enclaves. However, attacks on the operating system or other processes on the same host would still be possible.

To prevent attacks on the operating system or other processes, it would be necessary to partition the rest of the memory as well, *i.e.*, system-wide cache coloring [43]. Godfrey et al. [16] evaluated a coloring method for hypervisors by assigning every virtual machine a partition of the cache. They concluded that this method is only feasible for a small number of partitions. As the number of simultaneous enclaves is relatively limited by the available amount of SGX memory, enclave coloring can be applied to prevent cross-enclave attacks. Protecting enclaves from malicious applications or preventing malware inside enclaves is however not feasible using this method.

Heap Randomization. Our attack relies on the fact, that the used buffers for the multiplication are always at the same memory location. This is the case, as the used memory allocator (`dlmalloc`) has a deterministic best-fit strategy for moderate buffer sizes as used in RSA. Freeing a buffer and allocating it again will result in the same memory location for the re-allocated buffer.

We suggest randomizing the heap allocations for security relevant data such as the used buffers. A randomization of the addresses and thus cache sets bears two advantages. First, automatic cache set detection is not possible anymore, as the identified set will change for every run of the algorithm. Second, if more than one trace is required to reconstruct the key, heap randomization increases the number of required traces by multiple orders of magnitude, as the probability to measure the correct cache set by chance decreases.

Although not obvious at first glance, this method requires a certain amount of trust in the operating system. A malicious operating system could assign only pages mapping to certain cache sets to the enclave, similar to enclave coloring. Thus, the randomization is limited to only a subset of cache sets, increasing the probability for an attacker to measure the correct cache set.

Intel CAT. Recently, Intel introduced an instruction set extension called CAT (cache allocation technology) [24]. With Intel CAT it is possible to restrict CPU cores to one of the slices of the last-level cache and even to pin cache lines. Liu et al. [32] proposed a system that uses CAT to protect general purpose software and cryptographic algorithms. Their approach can be directly applied to protect against a malicious enclave. However, this approach does not allow to protect enclaves from an outside attacker.

7.3 Hardware Level

Combining Intel CAT with SGX. Instead of using Intel CAT on the operating system level it could also be used to protect enclaves on the hardware level. By changing the `eenter` instruction in a way that it implicitly activates CAT for this core, any cache sharing between SGX enclaves and the outside as well as co-located enclaves could be eliminated. Thus, SGX enclaves would be protected from outside attackers. Furthermore, it would protect co-located enclaves as well as the operating system and user programs against malicious enclaves.

Secure RAM. To fully mitigate cache- or DRAM-based side-channel attacks memory must not be shared among processes. We propose an additional fast, non-cachable secure memory element that resides inside the CPU.

The SGX driver can then provide an API to acquire the element for temporarily storing sensitive data. A cryptographic library could use this memory to execute code which depends on secret keys such as the square-and-multiply algorithm. Providing such a secure memory element per CPU core would even allow parallel execution of multiple enclaves.

Data from this element is only accessible by one program, thus cache attacks and DRAM-based attacks are not possible anymore. Moreover, if this secure memory is inside the CPU, it is infeasible for an attacker to mount physical attacks. It is unclear whether the Intel eDRAM implementation can already be instrumented as a secure memory to protect applications against cache attacks.

8 Conclusion

Intel claimed that SGX features impair side-channel attacks and recommends using SGX enclaves to protect cryptographic computations. Intel also claimed that enclaves cannot perform harmful operations.

In this paper, we demonstrated the first malware running in real SGX hardware enclaves. We demonstrated cross-enclave private key theft in an automated semi-synchronous end-to-end attack, despite all restrictions of SGX, e.g., no timers, no large pages, no physical addresses, and no shared memory. We developed a timing measurement technique with the highest resolution currently known for Intel CPUs, perfectly tailored to the hardware. We combined DRAM and cache side channels, to build a novel approach that recovers physical address bits without assumptions on the page size. We attack the RSA implementation of *mbedTLS*, which uses constant-time multiplication primitives. We extract 96 % of a 4096-bit RSA key from a single *Prime+Probe* trace and achieve full key recovery from only 11 traces.

Besides not fully preventing malicious enclaves, SGX provides protection features to conceal attack code. Even the most advanced detection mechanisms using performance counters cannot detect our malware. This unavoidably provides attackers with the ability to hide attacks as it eliminates the only known technique to detect cache side-channel attacks. We discussed multiple design issues in SGX and proposed countermeasures for future SGX versions.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments".

References

1. Anati, I., McKeen, F., Gueron, S., Huang, H., Johnson, S., Leslie-Hurd, R., Patil, H., Rozas, C.V., Shafi, H.: Intel Software Guard Extensions (Intel SGX) (2015), tutorial Slides presented at ICSA 2015
2. ARMmbed: Reduce mbed TLS memory and storage footprint. `https://tls.mbed.org/kb/how-to/reduce-mbedtls-memory-and-storage-footprint` (February 2016), retrieved on October 24, 2016
3. Arnaud, C., Fouque, P.A.: Timing attack against protected rsa-crt implementation used in polarssl. In: CT-RSA 2013 (2013)
4. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., et al.: Scone: Secure linux containers with intel sgx. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (2016)
5. Biham, E.: A fast new des implementation in software. In: International Workshop on Fast Software Encryption. pp. 260–272 (1997)
6. Blömer, J., May, A.: New partial key exposure attacks on rsa. In: Crypto'03 (2003)
7. Boneh, D., Durfee, G., Frankel, Y.: An attack on rsa given a small fraction of the private key bits. In: International Conference on the Theory and Application of Cryptology and Information Security (1998)
8. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.: Software grand exposure: SGX cache attacks are practical (2017), `http://arxiv.org/abs/1702.07521`
9. Chiappetta, M., Savas, E., Yilmaz, C.: Real time detection of cache-based side-channel attacks using hardware performance counters. Cryptology ePrint Archive, Report 2015/1034 (2015)
10. Costan, V., Devadas, S.: Intel sgx explained. Tech. rep., Cryptology ePrint Archive, Report 2016/086 (2016)
11. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
12. Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., Stolfo, S.: On the feasibility of online malware detection with performance counters. ACM SIGARCH Computer Architecture News 41(3), 559–570 (2013)
13. Docker: Amazon web services - docker. `https://docs.docker.com/machine/drivers/aws/` (2016)
14. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS) (2015)
15. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Tech. rep., Cryptology ePrint Archive, Report 2016/613, 2016. (2016)
16. Godfrey, M.M., Zulkernine, M.: Preventing cache-based side-channel attacks in a cloud environment. IEEE Transactions on Cloud Computing (Oct 2014)
17. Götzfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache attacks on intel sgx. In: Proceedings of the 10th European Workshop on Systems Security (EuroSec'17) (2017)
18. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA'16 (2016)

19. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA'16 (2016)
20. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium (2015)
21. Herath, N., Fogh, A.: These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In: Black Hat USA (2015)
22. Hund, R., Willems, C., Holz, T.: Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P'13 (2013)
23. Intel: Intel® 64 and IA-32 Architectures Optimization Reference Manual (2014)
24. Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide 253665 (2014)
25. Intel Corporation: Hardening Password Managers with Intel Software Guard Extensions: White Paper (2016)
26. Intel Corporation: Intel SGX: Debug, Production, Pre-release what's the difference? `https://software.intel.com/en-us/blogs/2016/01/07/intel-sgx-debug-production-prelease-whats-the-difference` (January 2016), retrieved on October 24, 2016
27. Intel Corporation: Intel Software Guard Extensions (Intel SGX). `https://software.intel.com/en-us/sgx` (2016), retrieved on November 7, 2016
28. Intel Corporation: Intel(R) Software Guard Extensions for Linux* OS. `https://github.com/01org/linux-sgx-driver` (2016), retrieved on November 11, 2016
29. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a minute! A fast, Cross-VM attack on AES. In: RAID'14 (2014)
30. Kocher, P.C.: Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: Crypto'96 (1996)
31. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium (2016)
32. Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., Lee, R.B.: Catalyst: Defeating last-level cache side channel attacks in cloud computing. In: IEEE International Symposium on High Performance Computer Architecture (HPCA'16) (2016)
33. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: S&P'15 (2015)
34. Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., Francillon, A.: Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID'15 (2015)
35. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Boano, C.A., Mangard, S., Römer, K.: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS'17 (2017)
36. Microsoft: Create a docker environment in azure using the docker vm extension. `https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-dockerextension/` (Oct 2016)
37. Moghimi, A., Irazoqui, G., Eisenbarth, T.: CacheZoom: How SGX Amplifies The Power of Cache Attacks. arXiv preprint arXiv:1703.06986 (2017)
38. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS'15 (2015)
39. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA 2006 (2006)
40. Payer, M.: HexPADS: a platform to detect "stealth" attacks. In: ESSoS'16 (2016)

41. Pereida García, C., Brumley, B.B., Yarom, Y.: Make sure dsa signing exponentiations really are constant-time. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (2016)
42. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium (2016)
43. Raj, H., Nathuji, R., Singh, A., England, P.: Resource Management for Isolation Enhanced Cloud Services. In: Proceedings of the 1st ACM Cloud Computing Security Workshop (CCSW'09). pp. 77–84 (2009)
44. Rutkowska, J.: Thoughts on Intel's upcoming Software Guard Extensions (Part 2). `http://theinvisiblethings.blogspot.co.at/2013/09/thoughts-on-intels-upcoming-software.html` (September 2013), retrieved on October 20, 2016
45. Schindler, W.: Exclusive exponent blinding may not suffice to prevent timing attacks on rsa. In: International Workshop on Cryptographic Hardware and Embedded Systems (2015)
46. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: Vc3: trustworthy data analytics in the cloud using sgx (2015)
47. Sudhakar, M., Kamala, R.V., Srinivas, M.: A bit-sliced, scalable and unified montgomery multiplier architecture for rsa and ecc. In: 2007 IFIP International Conference on Very Large Scale Integration. pp. 252–257 (2007)
48. Walter, C.D.: Longer keys may facilitate side channel attacks. In: International Workshop on Selected Areas in Cryptography (2003)
49. Wray, J.C.: An analysis of covert timing channels. Journal of Computer Security (1992)
50. Xu, Y., Cui, W., Peinado, M.: Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P'15 (May 2015)
51. Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium (2014)

Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features

Michael Schwarz[1], Daniel Gruss[1], Moritz Lipp[1], Clémentine Maurice[2],
Thomas Schuster[1], Anders Fogh[3], Stefan Mangard[1]
[1] Graz University of Technology, Austria
[2] CNRS, IRISA, France
[3] G DATA Advanced Analytics, Germany

ABSTRACT

Double-fetch bugs are a special type of race condition, where an unprivileged execution thread is able to change a memory location between the time-of-check and time-of-use of a privileged execution thread. If an unprivileged attacker changes the value at the right time, the privileged operation becomes inconsistent, leading to a change in control flow, and thus an escalation of privileges for the attacker. More severely, such double-fetch bugs can be introduced by the compiler, entirely invisible on the source-code level.

We propose novel techniques to efficiently detect, exploit, and eliminate double-fetch bugs. We demonstrate the first combination of state-of-the-art cache attacks with kernel-fuzzing techniques to allow fully automated identification of double fetches. We demonstrate the first fully automated reliable detection and exploitation of double-fetch bugs, making manual analysis as in previous work superfluous. We show that cache-based triggers outperform state-of-the-art exploitation techniques significantly, leading to an exploitation success rate of up to 97 %. Our modified fuzzer automatically detects double fetches and automatically narrows down this candidate set for double-fetch bugs to the exploitable ones. We present the first generic technique based on hardware transactional memory, to eliminate double-fetch bugs in a fully automated and transparent manner. We extend defensive programming techniques by retrofitting arbitrary code with automated double-fetch prevention, both in trusted execution environments as well as in syscalls, with a performance overhead below 1 %.

CCS CONCEPTS

• **Security and privacy → Operating systems security**;

ACM Reference Format:
Michael Schwarz[1], Daniel Gruss[1], Moritz Lipp[1], Clémentine Maurice[2], Thomas Schuster[1], Anders Fogh[3], Stefan Mangard[1]. 2018. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In *Proceedings of ACM Asia Conference on Computer and Communications Security (ASIACCS'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/http://dx.doi.org/10.1145/XXXXXX.XXXXXX

1 INTRODUCTION

The security of modern computer systems relies fundamentally on the security of the operating system kernel, providing strong isolation between processes. While kernels are increasingly hardened against various types of memory corruption attacks, race conditions are still a non-trivial problem. Syscalls are a common scenario in which the trusted kernel space has to interact with the untrusted user space, requiring sharing of memory locations between the two environments. Among possible bugs in this scenario are time-of-check-to-time-of-use bugs, where the kernel accesses a memory location twice, first to check the validity of the data and second to use it (double fetch) [65]. If such double fetches are exploitable, they are considered double-fetch bugs. The untrusted user space application can change the value between the two accesses and thus corrupt kernel memory and consequently escalate privileges. Double-fetch bugs can not only be introduced at the source-code level but also by compilers, entirely invisible for the programmer and any source-code-level analysis technique [5]. Recent research has found a significant amount of double fetches in the kernel through static analysis [70], and memory access tracing through full CPU emulation [38]. Both works had to manually determine for every double fetch, whether it is a double-fetch bug.

Double fetches have the property that the data is fetched twice from memory. If the data is already in the cache (*cache hit*), the data is fetched from the cache, if the data is not in the cache (*cache miss*), it is fetched from main memory into the cache. Differences between fetches from cache and memory are the basis for so-called cache attacks, such as Flush+Reload [56, 77], which obtain secret information by observing memory accesses [21]. Instead of exploiting the cache side channel for obtaining secret information, we utilize it to detect double fetches.

In this paper, we show how to efficiently and automatically detect, exploit, and eliminate double-fetch bugs, with two new approaches: DECAF and DropIt.

DECAF is a double-fetch-exposing cache-guided augmentation for fuzzers, which automatically *detects* and *exploits* real-world double-fetch bugs in a two-phase process. In the profiling phase, DECAF relies on cache side channel information to detect whenever the kernel accesses a syscall parameter. Using this novel technique, DECAF is able to detect whether a parameter is fetched multiple times, generating a candidate set containing double fetches, *i.e.*, some of which are potential double-fetch bugs. In the exploitation phase, DECAF uses a cache-based trigger signal to flip values while fuzzing syscalls from the candidate set, to trigger actual double-fetch bugs. In contrast to previous purely probability-based

ASIACCS'18, June 4–8, 2018, Songdo, Incheon, Korea
© 2018 Copyright held by the owner/author(s).
ACM ISBN ISBN 978-1-4503-5576-6/18/06.
https://doi.org/http://dx.doi.org/10.1145/XXXXXX.XXXXXX

approaches, cache-based trigger signals enable deterministic double-fetch-bug exploitation. Our automated exploitation exceeds state-of-the-art techniques, where checking the double-fetch candidate set for actual double-fetch bugs is tedious manual work. We show that DECAF can also be applied to trusted execution environments, e.g., ARM TrustZone and Intel SGX.

DropIt is a protection mechanism to *eliminate* double-fetch bugs. DropIt uses hardware transactional memory to efficiently drop the current execution state in case of a concurrent modification. Hence, double-fetch bugs are automatically reduced to ordinary non-exploitable double fetches. In case user-controlled memory locations are modified, DropIt continues the execution from the last consistent state. Applying DropIt to syscalls induces no performance overhead on arbitrary computations running in other threads and only a negligible performance overhead of 0.8 % on the process executing the protected syscall. We show that DropIt can also be applied to trusted execution environments, e.g., ARM TrustZone and Intel SGX.

Contributions. We make the following contributions:

1. We are the first to combine state-of-the-art cache attacks with kernel-fuzzing techniques to build DECAF, a generic double-fetch-exposing cache-guided augmentation for fuzzers.
2. Using DECAF, we are the first to show fully automated reliable detection and exploitation of double-fetch bugs, making manual analysis as in previous work superfluous.
3. We outperform state-of-the-art exploitation techniques significantly, with an exploitation success rate of up to 97 %.
4. We present DropIt, the first generic technique to eliminate double-fetch bugs in a fully automated manner, facilitating newfound effects of hardware transactional memory on double-fetch bugs. DropIt has a negligible performance overhead of 0.8 % on protected syscalls.
5. We show that DECAF can also fuzz trusted execution environments in a fully automated manner. We observe strong synergies between Intel SGX and DropIt, enabling efficient preventative protection from double-fetch bugs.

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background on cache attacks, race conditions, and kernel fuzzing. In Section 3, we discuss the building blocks for finding and eliminating double-fetch bugs. We present the profiling phase of DECAF in Section 4 and the exploitation phase of DECAF in Section 5. In Section 6, we show how hardware transactional memory can be used to eliminate all double-fetch bugs generically. In Section 7 we discuss the results of we obtained by instantiating DECAF. We conclude in Section 8.

2 BACKGROUND

2.1 Fuzzing

Fuzzing describes the process of testing applications with randomized input to find vulnerabilities.

The term "fuzzing" was coined 1988 by Miller [50], and later on extended to an automated approach for testing the reliability of several user-space programs on Linux [51], Windows [18] and Mac OS [49]. There is an immense number of works exploring user space fuzzing with different forms of feedback [12, 14, 19, 22–25, 32, 39, 61, 67]. However, these are not applicable to this work, as we focus on fuzzing the kernel and trusted execution environments.

Fuzzing is not limited to testing user-space applications, but it is also, to a much smaller extent, used to test the reliability of operating systems. Regular user space fuzzers cannot be used here, but a smaller number of tools have been developed to apply fuzzy testing to operating system interfaces. Carrette [9] developed the tool CrashMe that tests the robustness of operating systems by trying to execute random data streams as instructions. Mendoncca et al. [48] and Jodeit et al. [36] demonstrate that fuzzing drivers via the hardware level is another possibility to attack an operating system. Other operating system interfaces that can be fuzzed include the file system [7] and the virtual machine interface [20, 46].

The syscall interface is a trust boundary between the trusted kernel, running with the highest privileges, and the unprivileged user space. Bugs in this interface can be exploited to escalate privileges. Koopman et al. [40] were among the first to test random inputs to syscalls. Modern syscall fuzzers, such as Trinity [37] or syzkaller [69], test most syscalls with semi-intelligent arguments instead of totally random inputs. In contrast to these generic tools, Weaver et al. [71] developed perf_fuzzer, which uses domain knowledge to fuzz only the performance monitoring syscalls.

2.2 Flush+Reload

Flush+Reload is a side-channel attack exploiting the difference in access times between CPU caches and main memory. Yarom and Falkner [77] presented Flush+Reload as an improvement over the cache attack by Gullasch et al. [31]. Flush+Reload relies on shared memory between the attacker and the victim and works as follows:

1. Establish a shared memory region with the victim (e.g., by mapping the victim binary into the address space).
2. Flush one line of the shared memory from the cache.
3. Schedule the victim process.
4. Measure the access time to the flushed cache line.

If the victim accesses the cache line while being scheduled, it is again cached. When measuring the access time, the attacker can distinguish whether the data is cached or not and thus infer whether the victim accessed it. As Flush+Reload works on cache line granularity (usually 64 B), fine-grained attacks are possible. The probability of false positives is very low with Flush+Reload, as cache hits cannot be caused by different programs and prefetching can be avoided. Gruss et al. [28] reported extraordinarily high accuracies, above 99 %, for the Flush+Reload side channel, making it a viable choice for a wide range of applications.

2.3 Double Fetches and Double-Fetch Bugs

In a scenario where shared memory is accessed multiple times, the CPU may fetch it multiple times into a register. This is commonly known as a **double fetch**. Double fetches occur when the kernel accesses data provided by the user multiple times, which is often unavoidable. If proper checks are done, ensuring that a change in the data during the fetches is correctly handled, double fetches are **non-exploitable** valid constructs.

A **double-fetch bug** is a time-of-check-to-time-of-use race condition, which is **exploitable** by changing the data in the shared memory between two accesses. Double-fetch bugs are a subset of double fetches. A double fetch is a double-fetch bug, if and only if it can be exploited by concurrent modification of the data. For example, if a syscall expects a string and first checks the length of the string before copying it to the kernel, an attacker could change the string to a longer string after the check, causing a buffer overflow in the kernel. This can lead to code execution within the kernel.

Wang et al. [70] used Coccinelle, a transformation and matching engine for C code, to find double fetches. With this static pattern-based approach, they identified 90 double fetches inside the Linux kernel. However, their work incurred several days of manual analysis of these 90 double fetches, identifying only 3 exploitable double-fetch bugs. A further limitation of their work is that double-fetch bugs not matching the implemented patterns, cannot be detected. Xu et al. [75] used static code analysis in combination with symbolic checking to identify 23 new bugs in Linux. Again, double-fetch bugs not matching their formal definition are not identified.

Not all double fetches, and thus not all double-fetch bugs, can be found using static code analysis. Blanchou [5] demonstrated that especially in lock-free code, compilers can introduce double fetches that are not present in the code. Even worse, these compiler-introduced double fetches can become double-fetch bugs in certain scenarios (e.g., CVE-2015-8550). Jurczyk et al. [38] presented a dynamic approach for finding double fetches. They used a full CPU emulator to run Windows and log all memory accesses. Note that this requires significant computation and storage resources, as just booting Windows already consumes 15 hours of time, resulting in a log file of more than 100 GB [38]. In the memory access log, they searched for a distinctive double-fetch pattern, e.g., two reads of the same user-space address within a short time frame. They identified 89 double fetches in Windows 7 and Windows 8. However, their work also required manual analysis, in which they found that only 2 out of around 100 unique double fetches were exploitable double-fetch bugs. Again, if a double-fetch bug does not match the implemented double-fetch pattern, it is not detected. In summary, we find that all techniques for double-fetch bug detection are probabilistic and hence incomplete.

2.3.1 Race Condition Detection. Besides research on double fetches and double-fetch bugs, there has been a significant amount of research on race condition detection in general. Static analysis of source code and dynamic runtime analysis have been used to find data race conditions in multithreaded applications. Savage et al. [62] described the Lockset algorithm. Their tool, Eraser, dynamically detects race conditions in multithreaded programs. Poznianksy et al. [59, 60] extended their work to detect race conditions in multithreaded C++ programs on-the-fly. Yu et al. [79] described RaceTrack, an adaptive detection algorithm that reports suspicious activity patterns. These algorithms have been improved and made more efficient by using more lightweight data structures [17] or combining various approaches [74].

While these tools can be applied to user space programs, they are not designed to detect race conditions in the kernel space. Erickson et al. [15] utilized breakpoints and watchpoints on memory accesses to detect data races in the Windows kernel. With RaceHound [54], the same idea has been implemented for the Linux kernel. The SLAM [3] project uses symbolic model checking, program analysis, and theorem proving, to verify whether a driver correctly interacts with the operating system. Schwarz et al. [63] utilized software model checking to detect security violations in a Linux distribution.

More closely related to double-fetch bugs, other time-of-check-to-time-of-use bugs exist. By changing the content of a memory location that is passed to the operating system, the content of a file could be altered after a validity check [4, 8, 72]. Especially time-of-check-to-time-of-use bugs in the file system are well-studied, and several solutions have been proposed [13, 42, 57, 58, 68].

2.4 Hardware Transactional Memory

Hardware transactional memory is designed for optimizing synchronization primitives [16, 78]. Any changes performed inside a transaction are not visible to the outside before the transaction succeeds. The processor speculatively lets a thread perform a sequence of operations inside a transaction. Unless there is a conflict due to a concurrent modification of a data value, the transaction succeeds. However, if a conflict occurs before the transaction is completed (e.g., a concurrent write access), the transaction aborts. In this case, all changes that have been performed in the transaction are discarded, and the previous state is recovered. These fundamental properties of hardware transactional memory imply that once a value is read in a transaction, the value cannot be changed from outside the transaction anymore for the time of the transaction.

Intel TSX is a hardware transactional memory implementation with cache line granularity. It is available on several CPUs starting with the Haswell microarchitecture. Intel TSX maintains a read set which is limited to the size of the L3 cache and a write set limited to the size of the L1 cache [26, 35, 45, 80]. A cache line is automatically added to the read set when it is read inside a transaction, and it is automatically added to the write set when it is modified inside a transaction. Modifications to any memory in the read set or write set from other threads cause the transaction to abort.

Previous work has investigated whether hardware transactional memory can be instrumented for security features. Guan et al. [30] proposed to protect cryptographic keys by only decrypting them within TSX transactions. As the keys are never written to DRAM in an unencrypted form, they cannot be read from memory even by a physical attacker probing the DRAM bus. Kuvaiskii et al. [41] proposed to use TSX to detect hardware faults and roll-back the system state in case a fault occurred. Shih et al. [66] proposed to exploit the fact that TSX transactions abort if a page fault occurred for a memory access to prevent controlled-channel attacks [76] in cloud scenarios. Chen et al. [10] implemented a counting thread protected by TSX to detect controlled-channel attacks in SGX enclaves. Gruss et al. [29] demonstrated that TSX can be used to protect against cache side-channel attacks in the cloud.

Shih et al. [66] and Gruss et al. [29] observed that Intel TSX has several practical limitations. One observation is that executed code is not considered transactional memory, *i.e.*, virtually unlimited amount of code can be executed in a transaction. To evade the limitations caused by the L1 and L3 cache sizes, Shih et al. [66] and Gruss et al. [29] split transactions that might be memory-intense into multiple smaller transactions.

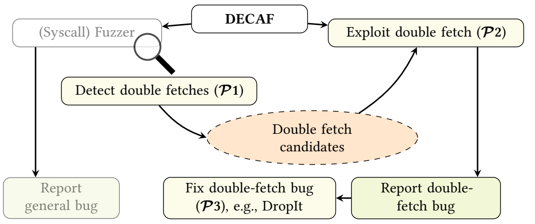

DECAF
(Syscall) Fuzzer
Exploit double fetch ($\mathcal{P}2$)
Detect double fetches ($\mathcal{P}1$)
Double fetch candidates
Report general bug
Fix double-fetch bug ($\mathcal{P}3$), e.g., DropIt
Report double-fetch bug

Figure 1: **An overview of the framework. Detecting (Primitive $\mathcal{P}1$) and exploiting (Primitive $\mathcal{P}2$) double fetches runs in parallel to the syscall fuzzer. Reported double-fetch bugs can be eliminated (Primitive $\mathcal{P}3$) after the fuzzing process.**

3 BUILDING BLOCKS TO DETECT, EXPLOIT, AND ELIMINATE DOUBLE-FETCH BUGS

In this section, we present building blocks for detecting double fetches, exploiting double-fetch bugs, and eliminating double-fetch bugs. These building blocks are the base for DECAF and DropIt.

We identified three primitives, illustrated in Figure 1, for which we propose novel techniques in this paper:

- **$\mathcal{P}1$**: Detecting double fetches via the Flush+Reload side channel.
- **$\mathcal{P}2$**: Distinguishing (exploitable) double-fetch bugs from (non-exploitable) double fetches by validating their exploitability by automatically exploiting double-fetch bugs.
- **$\mathcal{P}3$**: Eliminating (exploitable) double-fetch bugs by using hardware transactional memory.

In Section 4, we propose a novel, fully automated technique to detect double fetches ($\mathcal{P}1$) using a multi-threaded Flush+Reload cache side-channel attack. Our technique complements other work on double-fetch bug detection [38, 70] as it covers scenarios which lead to false positives and false negatives in other detection methods. Although relying on a side channel may seem unusual, this approach has certain advantages over state-of-the-art techniques, such as memory access tracing [38] or static code analysis [70]. We do not need any model of what constitutes a double fetch in terms of memory access traces or static code patterns. Hence, we can detect any double fetch regardless of any double fetch model.

Wang et al. [70] identified as limitations of their initial approach that false positives occur if a pointer is changed between two fetches and memory accesses, in fact, go to different locations or if user-space fetches occur in loops. Furthermore, false negatives occur if multiple pointers point to the same memory location (pointer aliasing) or if memory is addressed through different types (type conversion), or if an element is fetched separately from the corresponding pointer and memory. With a refined approach, they reduced the false positive rate from more than 98 % to only 94 %, *i.e.*, 6 % of the detected situations turned out to be actual double-fetch bugs in the manual analysis. Wang et al. [70] reported that it took an expert "only a few days" to analyze them. In contrast, our Flush+Reload-based approach is oblivious to language-level structures. The Flush+Reload-trigger only depends on actual accesses to the same memory location, irrespective of any programming constructs. Hence, we inherently bypass the problems of the approach of Wang et al. [70] by design.

Our technique does not replace existing tools, which are either slow [38] or limited by static code analysis [70] and require manual analysis. Instead, we complement previous approaches by utilizing a side channel, allowing fully automatic detection of double-fetch bugs, including those that previous approaches may miss.

In Section 5, we propose a novel technique to automatically determine whether a double fetch found using **$\mathcal{P}1$** is an (exploitable) double-fetch bug ($\mathcal{P}2$). State-of-the-art techniques are only capable of automatically detecting double fetches using either dynamic [38] or static [70] code analysis, but cannot determine whether a found double fetch is an exploitable double-fetch bug. The double fetches found using these techniques still require manual analysis to check whether they are valid constructs or exploitable double-fetch bugs. We close this gap by automatically testing whether double fetches are exploitable double-fetch bugs ($\mathcal{P}2$), eliminating the need for manual analysis. Again, this technique relies on a cache side channel to trigger a change of the double-fetched value between the two fetches ($\mathcal{P}2$). This is not possible with previous techniques [38, 70].

As the first automated technique, we present DECAF, a double-fetch-exposing cache-guided augmentation for fuzzers, leveraging **$\mathcal{P}1$** and **$\mathcal{P}2$** in parallel to regular fuzzing. This allows us to automatically detect double fetches in the kernel and to automatically narrow them down to the exploitable double-fetch bugs (cf. Section 5), as opposed to previous techniques [38, 70] which incurred several days of manual analysis work by an expert to distinguish double-fetch bugs from double fetches. Similar to previous approaches [38, 70], which inherently could not detect all double-fetch bugs in the analyzed code base, our approach is also probabilistic and might not detect all double-fetch bugs. However, due to their different underlying techniques, the previous approaches and ours complement each other.

In Section 6, we present a novel method to simplify the elimination of detected double-fetch bugs ($\mathcal{P}3$). We observe previously unknown interactions between double-fetch bugs and hardware transactional memory. Utilizing these effects, **$\mathcal{P}3$** can protect code without requiring to identify the actual cause of a double-fetch bug. Moreover, **$\mathcal{P}3$** can even be applied as a preventative measure to protect critical code.

As a practical implementation of **$\mathcal{P}3$**, we built DropIt, an open-source [1] instantiation of **$\mathcal{P}3$** based on Intel TSX. We implemented DropIt as a library, which eliminates double-fetch bugs with as few as 3 additional lines of code. We show that DropIt has the same effect as rewriting the code to eliminate the double fetch. Furthermore, DropIt can automatically and transparently eliminate double-fetch bugs in trusted execution environments such as Intel SGX, in both desktop and cloud environments.

4 DETECTING DOUBLE FETCHES

We propose a novel dynamic approach to detect double fetches based on their cache access pattern ($\mathcal{P}1$, cf. Section 3). The main idea is to monitor the cache access pattern of syscall arguments of a certain type, *i.e.*, pointers or structures containing pointers. These pointers may be accessed multiple times by the kernel and, hence, a second thread can change the content. Other arguments

[1] The source can be found at `https://www.github.com/IAIK/libdropit`.

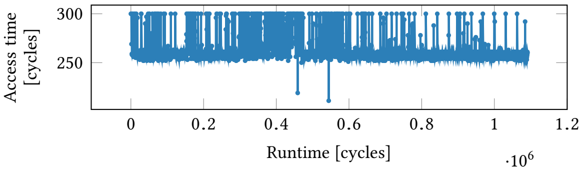


Figure 2: **Flush+Reload timing trace for a syscall with a double fetch. The two downward peaks show when the kernel accessed the argument.**

that are statically copied or passed by value, and consequently are not accessed multiple times, cannot lead to double fetches.

To monitor access to potentially vulnerable function arguments, we mount a Flush+Reload attack on each argument in dedicated monitoring threads. A monitoring thread continuously flushes and reloads the memory location referenced by the function argument. As soon as the kernel accesses the function argument, the data is loaded into the cache. In this case, the Flush+Reload attack in the corresponding monitoring thread reports a *cache hit.*

Figure 2 shows a trace generated by a monitoring thread. The trace contains the access time in cycles for the memory location referenced by the function argument. If the memory is accessed twice, *i.e.*, a double fetch, we can see a second cache hit, as shown in Figure 2. This provides us with primitive $\mathcal{P}\mathbf{1}$.

4.1 Classification of Multiple Cache Hits

Multiple cache hits within one trace usually correspond to multiple fetches. However, there are rare cases where this is not the case. To entirely eliminate spurious cache hits from prefetching, we simply disabled the prefetcher in software through MSR `0x1A4` and allocated memory on different pages to avoid spatial prefetching. Note that this does not have any effect on the overall system stability and only a small performance impact. We want to discuss two other factors influencing the cache access pattern in more detail.

Size of data type. Depending on the size of the data type, there are differences in the cache access pattern. If the data fills exactly one cache line, accesses to the cache line are clearly seen in the cache access pattern. There are no false positives due to unrelated data in the same cache set, and every access to the referenced memory is visible in the cache access pattern.

To avoid false positives if the data size is smaller than a cache line (*i.e.*, 64 B), we allocate memory chunks with a multiple of the page size, ensuring that dynamically allocated memory never shares one cache line. Hence, accesses to unrelated data (*i.e.*, separate allocations) do not introduce any false positives, as they are never stored in the same cache line. Thus, false positives are only detected if the cache line contains either multiple parameters, local variables or other members of the same structure.

Parameter reuse. With call-by-reference, one parameter of a function can be used both as input and output, e.g., in functions working in-place on a given buffer. Using Flush+Reload, we cannot distinguish whether a cache hit is due to a read of or write to the memory. Thus, we can only observe multiple cache hits without knowing whether they are both caused by a memory read access or by other activity on the same cache line.

4.2 Probability of Detecting a Double Fetch

The actual detection rate of a double fetch depends on the time between two accesses. Each Flush+Reload cycle consists of flushing the memory from the cache and measuring the access time to this memory location afterwards. Such a cycle takes on average 298 cycles on an Intel i7-6700K. Thus, to detect a double fetch, the time between the two memory accesses has to be at least two Flush+Reload cycles, *i.e.*, 596 CPU cycles.

We obtain the exact same results when testing a double fetch in kernel space as in user space. Also, due to the high noise-resistance of Flush+Reload (cf. Section 2), interrupts, context switches, and other system activity have an entirely negligible effect on the result. With the minimum distance of 596 CPU cycles, we can already detect double fetches if the scheduling is optimal for both applications. The further the memory fetches are apart, the higher the probability of detecting the double fetch. The probability of detecting double fetches increases monotonically with the time between the fetches, making it quite immune to interrupts such as scheduling. If the double fetches are at least 3000 CPU cycles apart, we almost always detect such a double fetch. In the real-world double-fetch bugs we examined, the double fetches were always significantly more than 3000 CPU cycles apart. Figure 9 (Appendix A) shows the relation between the detection probability and the time between the memory accesses, empirically determined on an Intel i7-6700K.

On a Raspberry Pi 3 with an ARMv8 1.2 GHz Broadcom BCM2837 CPU, a Flush+Reload cycle takes 250 cycles on average. Hence, the double fetches must be at least 500 cycles apart to be detectable with a high probability.

4.3 Automatically Finding Affected Syscalls

Using our primitive $\mathcal{P}\mathbf{1}$, we can already automatically and reliably detect whether a double fetch occurs for a particular function parameter. This is the first building block of DECAF. DECAF is a two-phase process, consisting of a profiling phase which finds double fetches and an exploitation phase narrowing down the set of double fetches to only double-fetch bugs. We will now discuss how DECAF augments existing fuzzers to discover double fetches within operating system kernels fully automatically.

To test a wide range of syscalls and their parameters, we instantiate DECAF with existing syscall fuzzers. For Linux, we retrofitted the well-known syscall fuzzer *Trinity* with our primitive $\mathcal{P}\mathbf{1}$. For Windows, we extended the basic *NtCall64* fuzzer to support semi-intelligent parameter selection similar to Trinity. Subsequently, we retrofitted our extended NtCall64 fuzzer with our primitive $\mathcal{P}\mathbf{1}$ as well. Thereby, we demonstrate that DECAF is a generic technique and does not depend on a specific fuzzer or operating system.

Our augmented and extended NtCall64 fuzzer, *NtCall64DECAF* works for double fetches and double-fetch bugs in proof-of-concept drivers. However, due to the severely limited coverage of the NtCall64 fuzzer, we did not include it in our evaluations. Instead, we focus on Linux only and leave retrofitting a good Windows syscall fuzzer with DECAF for future work.

In the profiling phase of DECAF, the augmented syscall fuzzer chooses a random syscall to test. The semi-intelligent parameter selection of the syscall fuzzer ensures that the syscall parameters are valid parameters in most cases. Hence, the syscall is executed and does not abort in the initial sanity checks.

Every syscall parameter that is either a pointer, a file or directory name, or an allocated buffer, can be monitored for double fetches. As Trinity already knows the data types of all syscall parameters, we can easily extend the main fuzzing routine. After Trinity selects a syscall to fuzz, it chooses the arguments to test with and starts a new process. Within this process, we spawn a Flush+Reload monitoring thread for every parameter that may potentially contain a double-fetch bug. The monitoring threads continuously flush the corresponding syscall parameter and measure the reload time. As soon as the parameter is accessed from kernel code, the monitoring thread measures a low access time. The threads report the number of detected accesses to the referenced memory after the syscall has been executed. These findings are logged, and simultaneously, all syscalls with double fetches are added to a candidate set for the interleaved exploitation phase. In Section 5, we additionally show how the second building block $\mathcal{P}$**2**, allows to automatically test whether such a double fetch is exploitable. Figure 8 (Appendix A) shows the process structure of our augmented version of Trinity, called *TrinityDECAF*.

4.4 Double-Fetch Detection for Black Boxes

The Flush+Reload-based detection method ($\mathcal{P}$**1**) is not limited to double fetches in operating system kernels. In general, we can apply the technique for all black boxes fulfilling the following criteria:

(1) Memory references can be passed to the black box.
(2) The referenced memory is (temporarily) shared between the black box and the host.
(3) It is possible to run code in parallel to the execution of the black box.

This generalization does not only apply to syscalls, but it also applies to trusted execution environments.

Trusted execution environments are particularly interesting targets for double fetch detection and double-fetch-bug exploitation. Trusted execution environments isolate programs from other user programs and the operating system kernel. These programs are often neither open source nor is the unencrypted binary available to the user. Thus, if the vendor did not test for double-fetch bugs, researchers not affiliated with the vendor have no possibility to scan for these vulnerabilities. Moreover, even the vendor might not be able to apply traditional double-fetch detection techniques, such as dynamic program analysis, if these tools are not available within the trusted execution environment.

Both Intel SGX [47] and ARM TrustZone [1] commonly share memory buffers between the host application and the trustlet running inside the trusted execution environment through their interfaces. Therefore, we can again utilize a Flush+Reload monitoring thread to detect double fetches by the trusted application ($\mathcal{P}$**1**).

5 EXPLOITING DOUBLE-FETCH BUGS

In this section, we detail the second building block of DECAF, primitive $\mathcal{P}$**2**, the base of the DECAF exploitation phase. It allows us to exploit any double fetch found via $\mathcal{P}$**1** (cf. Section 4) reliably and automatically. In contrast to state-of-the-art value flipping [38] (success probability 50 % or significantly lower), our exploitation phase has a success probability of 97 %. The success probability of value flipping is almost zero if multiple sanity checks are performed, whereas the success probability of $\mathcal{P}$**2** decreases only slightly.

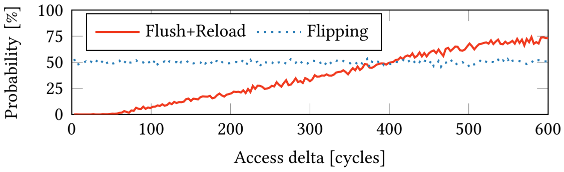


Figure 3: **The probability of successfully exploiting a double-fetch bug depending on the time between the accesses.**

5.1 Flush+Reload as a Trigger Signal

We propose to use Flush+Reload as a trigger signal to deterministically and reliably exploit double-fetch bugs. Indeed, Flush+Reload is a reliable approach to detect access to the memory, allowing us to flip the value immediately after an access. This combination of a trigger signal and targeted value flipping forms primitive $\mathcal{P}$**2**.

The idea of the double-fetch-bug exploitation ($\mathcal{P}$**2**) is therefore similar to the double-fetch detection ($\mathcal{P}$**1**). As soon as one access to a parameter is detected, the value of the parameter is flipped to an invalid value. Just as the double-fetch detection (cf. Section 4), we can use a double-fetch trigger signal for every black box which uses memory references as parameters in the communication interface.

As shown in Figure 3, the exploitation phase can target double-fetch bugs with an even lower time delta between the accesses, than the double-fetch detection in the profiling phase (cf. Section 4). The reason is that only the first access has to be detected and changing the value is significantly faster than a full Flush+Reload cycle. Thus, it is even possible to exploit double fetches where the time between them is already too short to detect them. Consequently, every double fetch detected in the profiling phase can clearly be tested for exploitability using $\mathcal{P}$**2** in the exploitation phase.

As a fast alternative to Flush+Reload, Flush+Flush [28] could be used. Although Flush+Flush is significantly faster than Flush+Reload, Flush+Reload is usually the better choice as it has less noise.

5.2 Automated Syscall Exploitation

With the primitive $\mathcal{P}$**2** from Section 5.1, we add the second building block to DECAF, to not only detect double fetches but also to immediately exploit them. This has the advantage that exploitable double-fetch bugs can be found without human interaction, as the automated exploitation leads to evident errors and system crashes. As described in Section 4, DECAF does not only report the double fetches but also adds them to a candidate set for double-fetch bug testing. If a candidate is added to this set, the double-fetch bug test ($\mathcal{P}$**2**) is immediately interleaved into the regular fuzzing process.

We randomly switch between four different methods to change the value: setting it to zero, flipping the least significant bit, incrementing the value, and replacing it by a random value. Setting a value to zero or a random value is useful to change pointers to invalid locations. Furthermore, it is also effective on string buffers as it can shorten the string, extend the string, or introduce invalid characters. Incrementing a value or flipping the least significant bit is especially useful if the referenced memory contains integers, as it might trigger off-by-one errors.

In summary, in the exploitation phase of DECAF, we reduce the double-fetch candidate set (obtained via $\mathcal{P}$**1**) to exploitable double-fetch bugs without any human interaction ($\mathcal{P}$**2**), complementing state-of-the-art techniques [38, 70]. The coverage of DECAF highly depends on the fuzzer used. Fuzzing is probabilistic and might not find every exploitable double fetch, but with growing coverage of fuzzers, the coverage of DECAF will automatically grow as well.

6 ELIMINATING DOUBLE-FETCH BUGS

In this section, we propose the first transparent and automated technique to entirely eliminate double-fetch bugs ($\mathcal{P}$**3**). We utilize previously unknown interactions between double-fetch bugs and hardware transactional memory. $\mathcal{P}$**3** protects code without requiring to identify the actual cause of a double-fetch bug and can even be applied as a preventative measure to protect critical code.

We present the DropIt library, an instantiation of $\mathcal{P}$**3** with Intel TSX. DropIt eliminates double-fetch bugs, having the same effect as rewriting the code to eliminate the double fetch. We also show its application to Intel SGX, a trusted execution environment that is particularly interesting in cloud scenarios.

6.1 Problems of State-of-the-Art Double-Fetch Elimination

Introducing double-fetch bugs in software happens easily, and they often stay undetected for many years. As shown recently, modern operating systems still contain a vast number of double fetches, some of which are exploitable double-fetch bugs [38, 70]. As shown in Section 4 and Section 5, identifying double-fetch bugs requires full code coverage, and before our work, a manual inspection of the detected double fetches. Even when double-fetch bugs are identified, they are usually not trivial to fix.

A simple example of a double-fetch bug is a syscall with a string argument of arbitrary length. The kernel requires two accesses to copy the string, first to retrieve the length of the string and allocate enough memory, and second, to copy the string to the kernel.

Writing this in a naïve way can lead to severe problems, such as unterminated strings of kernel buffer overflows. One approach is to use a retry logic, as shown in Algorithm 1 (Appendix B), as it used in the Linux kernel whenever user data of unknown length has to be copied to the kernel. Such methods increase the complexity and runtime of code, and they are hard to wrap into generic functions.

Finally, compilers can also introduce double fetches that are neither visible in the source code nor easily detectable, as they are usually within just a few cycles [5].

6.2 Generic Double-Fetch Bug Elimination

Eliminating double-fetch bugs is not equivalent to eliminating double fetches. Double fetches are valid constructs, as long as a change of the value is successfully detected, or it is not possible to change the value between two memory accesses. Thus, making a series of multiple fetches atomic is sufficient to eliminate double-fetch bugs, as there is only one operation from an attacker's view (see Section 2.4). Curiously, the concept of hardware transactional memory provides exactly this atomicity.

As also described in Section 2.4, transactional memory provides atomicity, consistency, and isolation [33]. Hence, by wrapping code possibly containing a double fetch within a hardware transaction, we can benefit from these properties. From the view of a different thread, the code is one atomic memory operation. If an attacker changes the referenced memory while the transaction is active, the transaction aborts and can be retried. As the retry logic is implemented in hardware and not simulated by software, the induced overhead is minimal, and the amount of code is drastically reduced.

In a nutshell, hardware transactional memory can be instrumented as a hardware implementation of software-based retry solutions discussed in Section 6.1. Thus, wrapping a double-fetch bug in a hardware transaction does not hide, but actually eliminates the bug ($\mathcal{P}$**3**). Similar to the software-based solution, our generic double-fetch bug elimination can be automatically applied in many scenarios, such as the interface between trust domains (e.g., `ECALL` in SGX). Naturally, solving a problem with hardware support is more efficient, and less error-prone, than a pure software solution.

In contrast to software-based retry solutions, our hardware-assisted solution ($\mathcal{P}$**3**) does not require any identification of the resource to be protected. For this reason, we can even prevent undetectable or yet undetected double-fetch bugs, regardless of whether they are introduced on the source level or by the compiler. As these interfaces are clearly defined, the double-fetch bug elimination can be applied in a transparent and fully automated manner.

6.3 Implementation of DropIt

To build DropIt, our instantiation of $\mathcal{P}$**3**, we had to rely on real-world hardware transactional memory, namely Intel TSX. Intel TSX comes with a series of imperfections, inevitably introducing practical limitations for security mechanisms, as observed in previous work [29] (cf. Section 2.4). However, as hardware transactional memory is exactly purposed to make multiple fetches from memory consistent, Intel TSX is sufficient for most real-world scenarios.

To eliminate double-fetch bugs, DropIt relies on the `XBEGIN` and `XEND` instructions of Intel TSX. `XBEGIN` specifies the start of a transaction as well as a fall-back path that is executed if the transaction aborts. `XEND` marks the successful completion of a transaction.

We find that on a typical Ubuntu Linux the kernel usually occupies less than 32 MB including all code, data, and heap used by the kernel and kernel modules. With an 8 MB L3 cache we could thus read or execute more than 20 % of the kernel without causing high abort rates [29] (cf. Section 2.4). In Section 7.4, we show that for practical use cases the abort rates are almost 0 % and our approach even improves the system call performance in several cases.

DropIt abstracts the transactional memory as well as the retry logic from the programmer. Hence, in contrast to existing software-based retry logic (cf. Section 6.1), e.g., in the Linux kernel, DropIt is mostly transparent to the programmer. To protect code, DropIt takes the number of automatic retries as a parameter as well as a fall-back function for the case that the transaction is never successful, *i.e.*, for the case of an ongoing attack. Hence, a programmer only has to add 3 lines of code to protect arbitrary code from double fetch exploitation. Listing 2 (Appendix E) shows an example how to protect the insecure `strcpy` function using DropIt. The solution with DropIt is clearly simpler than current state-of-the-art software-based retry logic (cf. Algorithm 1). Finally, replacing software-based retry logic by our hardware-assisted DropIt library can also improve the execution time of protected syscalls.

DropIt is implemented in standard C and does not have any dependencies. It can be used in user space, kernel space, and in trusted environments such as Intel SGX enclaves. If TSX is not available, DropIt immediately executes the fall-back function. This ensures that syscalls still work on older systems, while modern systems additionally benefit from possibly increased performance and elimination of yet unknown double-fetch bugs.

DropIt can be used for any code containing multiple fetches, regardless of whether they have been introduced on a source-code level or by the compiler. In case there is a particularly critical section in which a double fetch can cause harm, we can automatically protect it using DropIt. For example, this is possible for parts of syscalls that interact with the user space. As these parts are known to a compiler, a compiler can simply add the DropIt functions there.

DropIt is able to eliminate double-fetch bugs in most real-world scenarios. As Intel TSX is not an ideal implementation of hardware transactional memory, use of certain instructions in transactions is restricted, such as port I/O instructions [34]. However, double fetches are typically caused by string handling functions and do not rely on any restricted instructions. Especially in a trusted environment, such as Intel SGX enclaves, where I/O operations are not supported, all functions interacting with the host application can be automatically protected using DropIt. This is a particularly useful protection against an attacker in a cloud scenario, where an enclave containing an unknown double-fetch bug may be exposed to an attacker over an extended period of time.

7 EVALUATION

The evaluation consists of four parts. The first part evaluates DECAF ($\mathcal{P}1$ and $\mathcal{P}2$), the second part compares $\mathcal{P}2$ to state-of-the-art exploitation techniques, the third part evaluates $\mathcal{P}1$ on trusted execution environments, and the fourth part evaluates DropIt ($\mathcal{P}3$).

First, we demonstrate the proposed detection method using Flush+Reload. We evaluate the double-fetch detection of TrinityDECAF on both a recent Linux kernel 4.10 and an older Linux kernel 4.6 on Ubuntu 16.10 and discuss the results. We also evaluate the reliability of using Flush+Reload as a trigger in TrinityDECAF to exploit double-fetch bugs ($\mathcal{P}2$). On Linux 4.6, we show that TrinityDECAF successfully finds and exploits CVE-2016-6516.

Second, we compare our double-fetch bug exploitation technique ($\mathcal{P}2$) to state-of-the-art exploitation techniques. We show that $\mathcal{P}2$ outperforms value-flipping as well as a highly optimized exploit crafted explicitly for one double-fetch bug. This underlines that $\mathcal{P}2$ is both generic and extends the state of the art significantly.

Third, we evaluate the double-fetch detection ($\mathcal{P}1$) on trusted execution environments, *i.e.*, Intel SGX and ARM TrustZone. We show that despite the isolation of those environments, we can still use our techniques to detect double fetches.

Fourth, we demonstrate the effectiveness of DropIt, our double-fetch bug elimination method ($\mathcal{P}3$). We show that DropIt eliminates source-code-level double-fetch bugs with a very low overhead. Furthermore, we reproduce CVE-2015-8550, a compiler-introduced double-fetch bug. Based on this example we demonstrate that DropIt also eliminates double-fetch bugs which are not even visible in the source code. Finally, we measure the performance of DropIt protecting 26 syscalls in the Linux kernel, where TrinityDECAF reported double fetches.

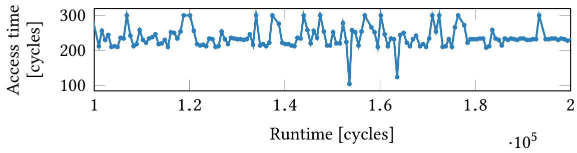


Figure 4: **The two memory accesses of the `FIDEDUPERANGE` `ioctl` in Linux kernel 4.5 to 4.7 can be clearly seen at around $1.5 \cdot 10^5$ and $1.6 \cdot 10^5$ cycles.**

7.1 Evaluation of DECAF

To evaluate DECAF, we analyze the double fetches and double-fetch bugs reported by TrinityDECAF. Our goal here is not to fuzz an excessive amount of time, but to demonstrate that DECAF constitutes a sensible and practical complement to existing techniques. Hence, we also used old and stable kernels where we did not expect to find new bugs, but validate our approach.

Reported Double-Fetch Bugs. Besides many double fetches TrinityDECAF reports in Linux kernel 4.6, it identifies one double-fetch bug which is already documented as CVE-2016-6516. It is a double-fetch bug in one of the `ioctl` calls. The syscall is used to share physical sections of two files if the content is identical.

When calling the syscall, the user provides a file descriptor for the source file as well as a starting offset and length within the source file. Furthermore, the syscall takes an arbitrary number of destination file descriptors with corresponding offsets and lengths. The kernel checks whether the given destination sections are identical to the source section and if this is the case, frees the sections and maps the source section into the destination file.

As the function allows for an arbitrary number of destination files, the user has to supply the number of provided destination files. This number is used to determine the amount of memory required to allocate. Listing 1 (Appendix C) shows the corresponding code from the Linux kernel. Changing the number between the allocation and the actual access to the data structure leads to a kernel heap-buffer overflow. Such an overflow can lead to a crash of the kernel or even worse to a privilege escalation.

Trinity already has rudimentary support for the `ioctl` syscall, which we extended with semi-intelligent defaults for parameter selection. Consequently, while Trinity does not find CVE-2016-6516, TrinityDECAF indeed successfully detects this double fetch in the profiling phase. Figure 4 shows a cache trace while calling the vulnerable function on Ubuntu 16.04 running an affected kernel 4.6. Although the time between the two accesses is only 10 000 cycles (approximately 2.5 µs on our i7-6700K test machine), we can clearly detect the two memory accesses.

When, in the exploitation phase, the monitoring thread changes the value to a higher value (cf. Section 5.2) exceeding the actual number of provided file descriptors, the kernel iterates out-of-bounds, as the number of file descriptors does not match the actual number of file descriptors anymore. This out-of-bounds access to the heap buffer results in a denial-of-service of the kernel and thus a hard reboot is required. Consequently, the denial-of-service shows that the double fetch is an exploitable double-fetch bug.

This demonstrates that DECAF is a useful complement to state-of-the-art fuzzing techniques, allowing to automatically detect bugs that cannot be found with traditional fuzzing approaches.

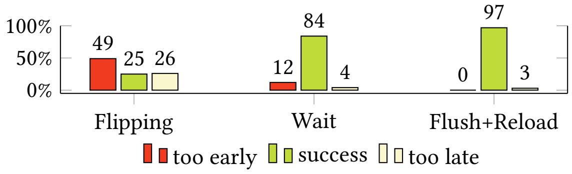


Figure 5: **Comparing three exploits for CVE-2016-6516. Our Flush+Reload-based trigger in TrinityDECAF succeeds in** 97 %**, outperforming the provided proof-of-concept** (84 %) **and the state-of-the-art method of value flipping** (25 %)**.**

Reported Double Fetches. Besides Linux kernel 4.6, we also tested TrinityDECAF on a recent Linux kernel 4.10. We let TrinityDECAF investigate all 64-bit syscalls (currently 295) without exceptions for one hour on an Intel i7-6700K. On average, every syscall was executed 8058 times. Due to the semi-intelligent parameter selection of TrinityDECAF, most syscalls are called with valid parameters. In our test run, 75.12 % of the syscalls executed successfully. Hence, on average, every syscall was successfully executed 6053 times, indicating a high code coverage for every syscall.

For every syscall parameter, TrinityDECAF displays a percentage of the calls where it detected a double fetch. Out of the 295 tested 64-bit syscalls, TrinityDECAF reported double fetches for 68 syscalls in Linux kernel 4.10. This is not surprising and in line with state-of-the-art work [70] which reported 90 double fetches in Linux, but only 33 in syscalls. For each of the reported syscalls, we investigated the respective implementation. Table 1 (Appendix A) shows a complete list of reported syscalls and the reason why TrinityDECAF detected a double fetch. We can group the reported syscalls into 5 major categories, explaining the detected double fetch.

- **Filenames.** Most syscalls handling filenames (or paths) are reported by TrinityDECAF. Many of them use `getname_flags` internally to copy a filename to a kernel buffer. This function checks whether the filename is already cached in the kernel, and copies it to the kernel if this is not the case, resulting in multiple accesses to the file name. The exploitation phase automatically filtered out all non-exploitable double fetches in this category.
- **Shared input/output parameters.** We found 5 syscalls which are reported by TrinityDECAF although they do not contain a double fetch. In these syscalls, one of the syscall parameters was used as input and output. As reads and writes are not distinguishable through the cache access pattern (cf. Section 4.1), these syscalls are filtered out automatically in the exploitation phase.
- **Strings of arbitrary length.** As with filenames, some syscalls expect strings from the user that do not have a fixed length. To safely copy such arbitrary length strings, some syscalls (e.g., `mount`) use an algorithm similar to Algorithm 1. Thus, the detected double fetch is due to the length check and the subsequent string copy. The exploitation phase automatically filtered out all non-exploitable double fetches in this category.
- **Sanity check.** Many syscalls check—either directly, or in a subroutine—whether the supplied argument is sane. There are sanity checks that check whether it is safe to access a user-space pointer before actually copying data from or to it. Such a check can also trigger a cache hit if the value was actually accessed. All correct sanity checks were automatically filtered out in the exploitation phase. The exploitation phase correctly identified the `ioctl` syscall in the Linux kernel 4.6, but also correctly filtered it out in Linux kernel 4.10.
- **Structure elements.** If a syscall has a structure as parameter, double fetches can be falsely detected if structure members fall into the same cache line (cf. Section 4.1). If members are either copied element-wise or neighboring members are simply accessed, TrinityDECAF will detect a double fetch although two different variables are accessed. Again, these false positives are filtered out in the exploitation phase.

Our evaluation showed that TrinityDECAF provides a sensible complement to existing double-fetch bug detection techniques. The fact that we found only 1 exploitable double-fetch bug in 68 double fetches is not surprising, and in line with previous work, e.g., Wang et al. [70] found 3 exploitable double-fetch bugs by manually inspecting 90 double fetches they found. However, it also shows that the coverage of DECAF highly depends on the fuzzer used to instantiate it. Future work may retrofit other fuzzers with DECAF, to extend the spectrum of bugs that the fuzzer covers and thereby also extend the coverage of DECAF. Furthermore, as Trinity is continuously extended, the coverage of TrinityDECAF grows automatically with the coverage of Trinity.

7.2 Evaluation of $\mathcal{P}2$

To evaluate $\mathcal{P}2$ in detail, we compare three different variants to exploit the double-fetch bug reported in CVE-2016-6516.

First, the provided exploit, which calls `ioctl` multiple times, always changing the affected variable after a slightly increased delay. Second, we use state-of-the-art value flipping to switch the affected variable as fast as possible between the valid and an invalid value. Third, the automated approach $\mathcal{P}2$, integrated into TrinityDECAF.

Figure 5 shows the success rate of 1000 executions of each of the three variants. Value flipping has by far the worst success rate, although in theory, it should have a success rate of approximately 50 %. In half of the cases, the value is flipped before the first access. Thus, the exploit fails, as the value is smaller at the next access. In the other cases, the probability to switch the value at the correct time is again 50 % resulting in an overall success rate of 25 %.

The original exploit is highly optimized for this specific vulnerability. It uses a trial-and-error busy wait with steadily increasing timeouts, which works surprisingly well, as there is sufficient time between the two accesses. Depending on the scheduling, the attacker sometimes sleeps too long (4 %) and sometimes too short

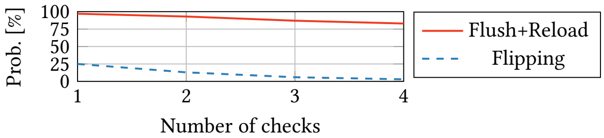


Figure 6: **The probability of double-fetch bug exploitation decreases with the number of sanity checks as it only succeeds if the value changes between the last two accesses.**

(12 %). Still, the busy wait outperforms the value flipping in this scenario, increasing the success probability from 25 % to 84 %.

Even though our Flush+Reload-based trigger (**𝒫2**) is generic and does not require fine-tuning of the sleep intervals, it has the highest success rate. There is no case where the value was changed too early, as there are no false positives with Flush+Reload in this scenario. Furthermore, as the time between the two memory accesses is long enough, we achieve an almost perfect success rate of 97 %. The remaining 3 %, where we do not trigger a change of the value, are caused by unfortunate scheduling of the application.

The success rate of value flipping drops significantly if the two values have to fulfill specific constraints, e.g., the value has to be higher on the second access. For example, if an application does not only fetch the value twice, but multiple times for sanity checking, the probability of successfully exploiting it using value flipping decreases exponentially.

Figure 6 shows the probability to exploit a double fetch similar to CVE-2016-6516, with additional fetches for sanity checks. To successfully exploit the vulnerability, the value has to be the same for all sanity checks and must be higher for the last access. Flipping the value between a valid and an invalid value decreases the chances by 50 % for every additional sanity check.

Our Flush+Reload-based method (**𝒫2**) does not suffer significantly from additional sanity checks. We can accurately trigger on the second to last access to change the value. The slightly decreased probability is only due to missed accesses.

7.3 Evaluation of 𝒫1 on Trusted Execution Environments

We evaluate **𝒫1** on trusted execution environments by successfully detecting double fetches in Intel SGX and ARM TrustZone.

Intel SGX. Intel SGX allows running code in secure enclaves without trusting the user or the operating system. A program inside an enclave is not accessible to the operating system due to hardware isolation provided by SGX. Weichbrodt et al. [73] showed that synchronization bugs, such as double fetches, inside SGX enclaves, can be exploited to hijack the control flow or bypass access control.

As it has been shown recently, enclaves leak information through the last-level cache, even to unprivileged user space applications, as they share the last-level cache with regular user space applications [6, 27, 53, 64]. SGX enclaves provide a communication interface using so-called `ecalls` and `ocalls`, similar to the syscall interface. Enclaves fulfill the properties of Section 4.4, and we can thus detect double fetches within enclaves, even without access to the binary. Therefore, we can apply our method to identify double fetches within SGX enclaves.

To test our Flush+Reload detection mechanism (**𝒫1**), we implemented a small enclave application. This application consists of only one `ecall`, which takes a memory reference as a parameter. As enclaves can access non-enclave memory, the user can simply allocate memory and provide the pointer to the enclave. The enclave accesses the memory once, idles a few thousand cycles and reaccesses the memory. Although the enclave should be isolated from other applications, the monitoring application can clearly detect the 2 cache hits. Figure 11 (Appendix D) shows the measurement of the Flush+Reload thread running outside the enclave on an Intel i5-6200U. Similarly, Appendix D evaluates **𝒫1** on ARM TrustZone.

7.4 Evaluation of DropIt

To evaluate our open-source library DropIt, as an instantiation of **𝒫3**, we investigate two real-world scenarios. In the first scenario, we demonstrate how DropIt eliminates a compiler-introduced real-world double-fetch bug in Xen (CVE-2015-8550). In the second scenario, we evaluate the effect of DropIt on Linux syscalls with double fetches. Our findings show that DropIt successfully eliminates all double-fetch bugs and can be used as a preventative measure to protect double fetches in syscalls generically.

Eliminating Compiler-Introduced Double-Fetch Bugs. As discussed in Section 2.3, compilers can also introduce double-fetch bugs. Especially switch statements are prone to double-fetch bugs if the variable is subject to a race condition [5, 52]. This is not an issue with the compiler, as the compiler is allowed to assume an atomic value for the switch condition [11]. We are aware of two scenarios where code generated by gcc contains a double-fetch bug.

If a switch is translated into a jump table with a default case, gcc generates two accesses to the switch variable. The first access checks whether the parameter is within the bounds of the jump table, the second access calculates the actual jump target. Thus, if the parameter changes between the accesses, a malicious user can divert the control flow of the program.

If the switch is implemented as multiple conditional jumps, the compiler is allowed to fetch the variable for every conditional jump. This leads to cases where the switch executes the default case as the variable changes while checking the conditions [52].

We evaluated DropIt on the real-world compiler-introduced double-fetch bug CVE-2015-8550. This vulnerability in Xen allowed arbitrary code execution due to a compiler-introduced double fetch within a switch statement. Note that such a switch statement is a common construct and can occur in any other kernel, e.g., Linux, or Windows, if a memory buffer is shared between user space and kernel space. Wrapping the switch statement using DropIt results in a clean and straightforward fix without relying on the compiler. With DropIt, any compiler-introduced switch-related double-fetch bug is successfully eliminated using only 3 lines of additional code.

To compare the overhead of traditional locking and DropIt, we implemented a minimal working example of a compiler-introduced double-fetch bug. Our example consists of a switch statement that has 5 different cases as well as a default case. The condition is a pointer which is subject to a race condition. The average execution time of the switch statement without any protection is 7.6 cycles.

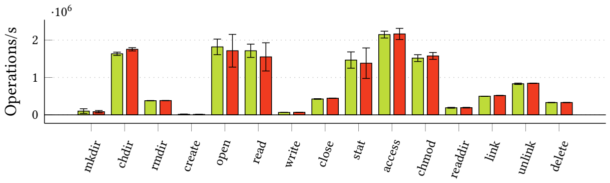


Figure 7: **The number of executed file operations per second of our re-implemented `getname_flags` using DropIt (green) does not significantly differ from the version in the vanilla kernel (red) measured with the IOZone Fileops benchmark.**

Using a spinlock to protect the variable increased the average execution time to 83.7 cycles. DropIt achieved a higher performance than the traditional spinlock with an average execution time of 68.0 cycles. Thus, DropIt is not only easy to deploy but also achieves a better performance than traditional locking mechanisms.

Preventative Protection of Linux Syscalls. To show that DropIt provides an automated and transparent generic solution to eliminate double-fetch bugs, we also used DropIt in the Linux kernel. As discussed in Section 7.1, a majority of the double fetches we detected in the Linux kernel are due to the `getname_flags` function handling file names. We replaced this function with a straight-forward implementation protected by DropIt. With this small change, all double fetches previously reported in 26 syscalls were covered by DropIt, and thus all potential double-fetch bugs were eliminated.

To compare the performance of DropIt with the vanilla implementation, we executed 210 million file operations in both cases. All benchmarks were run on a bare metal kernel to reduce the impact of system noise. Figure 7 shows the result of the IOzone filesystem benchmark [55]. On average, the benchmarks show a 0.8 % performance degradation on the tested file operations that are affected by our kernel change. In some cases, DropIt even has a better performance than the vanilla implementation. We therefore conclude that DropIt has no perceptible performance impact. The variances in the tests are probably due to the underlying hardware, *i.e.*, the SSDs on which we performed the file operations.

Thus, DropIt provides a reliable and straightforward way to cope with double-fetch bugs. It is easily integrable into existing C projects and does not negatively influence the performance compared to state-of-the-art solutions. Furthermore, it even increases the performance compared to traditional locking mechanisms. DropIt in SGX performs even better, since many operations interrupting TSX transactions are forbidden in SGX enclaves anyway.

8 CONCLUSION

In this paper, we proposed novel techniques to efficiently detect, exploit, and eliminate double-fetch bugs. We presented the first combination of state-of-the-art cache attacks with kernel-fuzzing techniques. This allowed us to find double fetches in a fully automated way. Furthermore, we presented the first fully automated reliable detection and exploitation of double-fetch bugs. By combining these two primitives, we built DECAF, a system to automatically find and exploit double-fetch bugs. DECAF is the first method that makes manual analysis of double fetches as in previous work superfluous. We show that cache-based triggers, as we use in DECAF, outperform state-of-the-art exploitation techniques significantly, leading to an exploitation success rate of up to 97 %.

DECAF constitutes a sensible complement to existing double-fetch detection techniques. Future work may retrofit more fuzzers with DECAF, extending the spectrum of bugs covered by fuzzers. Hence, double-fetch bugs do not require separate detection tools anymore, but testing for these bugs can now be a part of regular fuzzing. With continuously growing coverage of fuzzers, the covered search space for potential double-fetch bugs grows as well.

With DropIt, we leverage a newfound interaction between hardware transactional memory and double fetches, to completely eliminate double-fetch bugs. Furthermore, we showed that DropIt can be used in a fully automated manner to harden Intel SGX enclaves such that double-fetch bugs cannot be exploited. Finally, our evaluation of DropIt in the Linux kernel showed that it can be applied to large systems with a negligible performance overhead below 1 %.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their feedback. This work has been supported by the Austrian Research Promotion Agency (FFG), the Styrian Business Promotion Agency (SFG), the Carinthian Economic Promotion Fund (KWF) under grant number 862235 (DeSSnet) and has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

REFERENCES

[1] Tiago Alves. 2004. Trustzone: Integrated hardware and software security. (2004).
[2] ARM Limited. 2012. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition.* ARM Limited.
[3] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. 2004. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *International Conference on Integrated Formal Methods.* Springer.
[4] Matt Bishop, Michael Dilger, et al. 1996. Checking for race conditions in file accesses. *Computing systems* 2, 2 (1996).
[5] Marc Blanchou. 2013. Shattering Illusions in Lock-Free Worlds – Compiler and Hardware behaviors in OSes and VMs. In *Black Hat Briefings.*
[6] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT.*
[7] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* (2008).
[8] Xiang Cai, Yuwei Gui, and Rob Johnson. 2009. Exploiting UNIX file-system races via algorithmic complexity attacks. In *S&P.*
[9] George J Carrette. 1996. CRASHME: Random input testing. (1996).
[10] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *AsiaCCS.*
[11] ANSI Technical Committee, ISO/IEC JTC 1 Working Group, et al. 1999. Rationale for international standard, Programming languages - C. (1999).
[12] Bogdan Copos and Praveen Murthy. 2015. Inputfinder: Reverse engineering closed binaries using hardware performance counters. In *5th Program Protection and Reverse Engineering Workshop.*
[13] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. 2001. RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities.. In *USENIX Security Symposium.*
[14] M Eddington. 2008. Peach fuzzer. (2008).
[15] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-race Detection for the Kernel. In *OSDI.*
[16] Cesare Ferri, Ruth Iris Bahar, Mirko Loghi, and Massimo Poncino. 2009. Energy-optimal synchronization primitives for single-chip multi-processors. In *19th ACM Great Lakes symposium on VLSI.*
[17] Cormac Flanagan and Stephen N. Freund. 2010. FastTrack: Efficient and Precise Dynamic Race Detection. *Commun. ACM* (2010).

[18] Justin E Forrester and Barton P Miller. 2000. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows System Symposium.*

[19] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *31st International Conference on Software Engineering.*

[20] Amaury Gauthier, Clément Mazin, Julien Iguchi-Cartigny, and Jean-Louis Lanet. 2011. Enhancing fuzzing technique for OKL4 syscalls testing. In *Sixth International Conference on Availability, Reliability and Security (ARES).*

[21] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* (2016).

[22] Patrice Godefroid. 2007. Random testing for security: blackbox vs. whitebox fuzzing. In *2nd International Workshop on Random Testing.*

[23] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, Vol. 43. 206–215.

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM Sigplan Notices*, Vol. 40. 213–223.

[25] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Queue* 10, 1 (2012), 20.

[26] Bhavishya Goel, Ruben Titos-Gil, Anurag Negi, Sally A McKee, and Per Stenstrom. 2014. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *IEEE IPDPS.*

[27] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *EuroSec.*

[28] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA.*

[29] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security Symposium.*

[30] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. 2015. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *S&P.*

[31] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P.*

[32] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations.. In *USENIX Security Symposium.*

[33] Tim Harris, James Larus, and Ravi Rajwar. 2010. Transactional memory. *Synthesis Lectures on Computer Architecture* 5, 1 (2010), 1–263.

[34] Intel. 2012. Intel Architecture Instruction Set Extensions Programming Reference. (2012).

[35] Intel. 2017. Intel 64 and IA-32 Architectures Optimization Reference Manual. (2017).

[36] Moritz Jodeit and Martin Johns. 2010. Usb device drivers: A stepping stone into your kernel. In *IEEE European Conference on Computer Network Defense.*

[37] Dave Jones. 2011. Trinity: A system call fuzzer. In *13th Ottawa Linux Symposium.*

[38] Mateusz Jurczyk, Gynvael Coldwind, et al. 2013. Identifying and exploiting windows kernel race conditions via memory access patterns. (2013).

[39] Ulf Kargén and Nahid Shahmehri. 2015. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *10th Joint Meeting on Foundations of Software Engineering.*

[40] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. 1997. Comparing operating systems using robustness benchmarks. In *16th Symposium on Reliable Distributed Systems.*

[41] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2016. HAFT: Hardware-assisted fault tolerance. In *EuroSys.*

[42] Kyung-Suk Lhee and Steve J Chapin. 2005. Detection of file-based race conditions. *International Journal of Information Security* 4, 1 (2005).

[43] Linaro. 2016. OP-TEE: Open Portable Trusted Execution Environment. (2016). https://www.op-tee.org/

[44] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium.*

[45] Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. 2014. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[46] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing system virtual machines. In *19th International Symposium on Software Testing and Analysis.*

[47] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution.. In *HASP@ ISCA.*

[48] Manuel Mendonça and Nuno Neves. 2008. Fuzzing wi-fi drivers to locate security vulnerabilities. In *IEEE EDCC.*

[49] Barton P Miller, Gregory Cooksey, and Fredrick Moore. 2006. An empirical study of the robustness of macos applications using random testing. In *1st International Workshop on Random Testing.*

[50] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* (1990).

[51] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services.* Technical Report. University of Wisconsin.

[52] MITRE. 2017. CWE-365: Race Condition in Switch. (2017). https://cwe.mitre.org/data/definitions/365.html

[53] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies The Power of Cache Attacks. *arXiv:1703.06986* (2017).

[54] Nikita Komarov. 2015. Racehound: Data race detector for Linux kernel modules. (2015). https://github.com/winnukem/racehound

[55] William D Norcott and Don Capps. 2016. IOzone filesystem benchmark. (2016). http://www.iozone.org/

[56] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA.*

[57] Mathias Payer and Thomas R Gross. 2012. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *ACM SIGPLAN Notices.*

[58] Donald E Porter, Owen S Hofmann, Christopher J Rossbach, Alexander Benn, and Emmett Witchel. 2009. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.*

[59] Eli Pozniansky and Assaf Schuster. 2003. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *PPoPP.*

[60] Eli Pozniansky and Assaf Schuster. 2007. MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles. *Concurr. Comput. : Pract. Exper.* (2007).

[61] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS.*

[62] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems* (1997).

[63] Benjamin Schwarz, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. 2005. Model checking an entire linux distribution for security violations. In *Computer Security Applications Conference, 21st Annual.*

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

[65] Fermin J Serna. 2008. MS08-061 : The case of the kernel mode double-fetch. (2008). https://blogs.technet.microsoft.com/srd/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/

[66] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS.*

[67] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS.*

[68] Prem Uppuluri, Uday Joshi, and Arnab Ray. 2005. Preventing race condition attacks on file-systems. In *ACM Symposium on Applied Computing.*

[69] Dmitry Vyukov. 2016. syzkaller - linux syscall fuzzer. (2016). https://github.com/google/syzkaller

[70] Pengfei Wang and Jens Krinke. 2017. How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. In *USENIX Security Symposium.*

[71] Vincent M Weaver and Dave Jones. 2015. *perf fuzzer: Targeted fuzzing of the perf_event_open() system call.* Technical Report. Technical Report, University of Maine.

[72] Jinpeng Wei and Calton Pu. 2005. TOCTTOU Vulnerabilities in UNIX-style File Systems: An Anatomical Study. In *FAST.*

[73] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *ESORICS.*

[74] Xinwei Xie and Jingling Xue. 2011. Acculock: Accurate and Efficient Detection of Data Races. In *CGO.*

[75] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *S&P.*

[76] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P.*

[77] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium.*

[78] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *International Conference on High Performance Computing, Networking, Storage and Analysis.*

[79] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP.*

[80] Georgios Zacharopoulos. 2015. Employing Hardware Transactional Memory in Prefetching for Energy Efficiency. Examensarbete, Uppsala Universitet. (2015).

A TRINITYDECAF AND DETECTED DOUBLE FETCHES

In this section, we show implementation details of TrinityDECAF (our augmented version of Trinity) as well as a complete list of syscalls reported by TrinityDECAF.

Figure 8 shows the process structure of our augmented version of Trinity, called *TrinityDECAF*. The syscall fuzzer Trinity is extended with one monitoring threads per syscall argument. Each of the monitoring threads mounts a Flush+Reload attack to detect double fetches (cf. Section 4.3).

Figure 9 shows the probability that TrinityDECAF detects a double fetch depending on the time between the two accesses to the memory (cf. Section 4.2)

Table 1 is a complete table of reported syscalls and the reason why TrinityDECAF detected a double fetch. The categories are discussed in detail in Section 7.1.

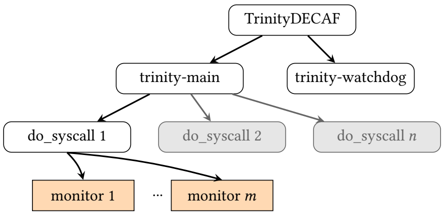


Figure 8: **The structure of TrinityDECAF with the Flush+ Reload monitoring threads for the syscall parameters.**

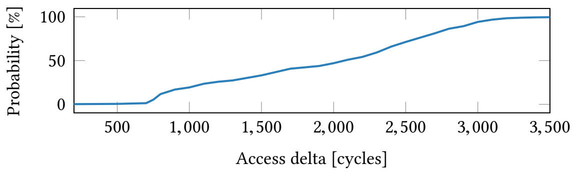


Figure 9: **The probability of detecting a double fetch depending on the time between the accesses.**

TABLE 1: **DOUBLE FETCHES FOUND BY TRINITYDECAF.**

Category	Syscall
Filenames	open, newstat, truncate, chdir, rename(at), mkdir(at), rmdir, creat, unlink, link, symlink(at), readlink(at), chmod, (l)chown, utime, mknod, statfs, chroot, quotactl, *xattr, fchmodat
Shared input/output	sendfile, adjtimex, io_setup, recvmmsg, sendmmsg
Strings	mount, memfd_create
Sanity check	sched_setparam, ioctl, sched_setaffinity, io_cancel, sched_setscheduler, futimesat, sysctl, settimeofday, gettimeofday
Structure elements	recvmsg, msgsnd, sigaltstack, utime

B SAFE STRING COPY

In this section, we show the pseudo-code of a standard algorithm used to safely copy an arbitrary-length string. Algorithm 1 first retrieves the length of the string, to allocate a buffer and copy the string up to this length. Then, it checks whether the string is terminated, and if not, retries again as the buffer was apparently changed before copying it.

A similar algorithm is used in the Linux kernel whenever user data of unknown length has to be copied to the kernel.

Algorithm 1: Safe string copy for arbitrary string lengths with software-based retry logic.

```
input  : string
copy:
len ← strlen(string);
buffer ← allocate(len + 1);
strncpy(buffer, string, len);
if not isNullTerminated(buffer) then
    free(buffer);
    goto copy; // or abort with error if too many retries
end
```

C CVE-2016-6516

CVE-2016-6516 is a double-fetch bug in an `ioctl` call used to share physical sections of two files if the content is identical. This deduplicates the identical section to save physical storage. On a write access, the identical section has to be copied to ensure that the changes are only visible within the changed file.

The user provides a file descriptor for the source file as well as a starting offset and length within the source file. Additionally, the syscall takes an arbitrary number of destination file descriptors including offsets and lengths. The kernel maps the source section into the destination file if the given destination sections are identical to the source section.

The function supports an arbitrary number of destination files. Thus, the user has to supply the number of provided destination files, so that the kernel can determine the required amount of memory to allocate. Listing 1 shows the corresponding code from the Linux kernel. If the number changes between the allocation and the actual access to the data structure, the kernel accesses the buffer out-of-bounds, leading to a heap-buffer overflow.

```
// first access of dest_count
if (get_user(count, &argp->dest_count)) { [...] }
// allocation based on dest_count
size = offsetof(struct file_dedupe_range __user,
  info[count]);
same = memdup_user(argp, size);
if (IS_ERR(same)) { [...] }
ret = vfs_dedupe_file_range(file, same);
// function accesses same->dest_count, not count
```

Listing 1: **The vulnerable `ioctl_file_ dedupe_range` function that was present in the Linux kernel from version 4.5 to 4.7. The `dest_count` member is accessed twice and can thus be changed between the accesses by a malicious user, leading to a kernel heap-buffer overflow.**

D ARM TRUSTZONE

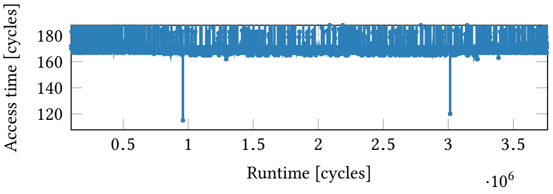


Figure 10: **A double fetch of a trustlet running inside the ARM TrustZone of a Raspberry Pi 3. The cache hits can be clearly seen at around** $0.96 \cdot 10^6$ **and** $3.01 \cdot 10^6$ **cycles as the access time drops from** >160 **cycles to** <120 **cycles.**

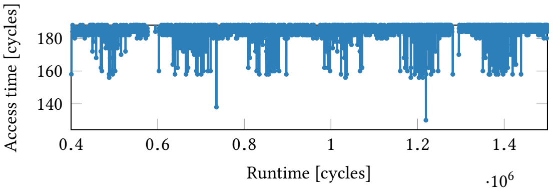


Figure 11: **Monitoring a double fetch inside an SGX enclave. The cache hits can be clearly seen at around** $0.75 \cdot 10^6$ **and** $1.21 \cdot 10^6$ **cycles as the access time drops from** >150 **cycles to** <140 **cycles.**

ARM TrustZone is a trusted execution environment for the ARM platform. The processor can either run in the normal world or the trusted world. As with Intel SGX, the worlds are isolated from each other using hardware mechanisms. Trustlets—applications running inside the secure world—provide a well-defined interface to normal world applications. This interface is accessed through a secure monitor call, similar to a syscall.

To use the ARM TrustZone, the normal-world operating system requires TrustZone support. Furthermore, a secure-world operating system has to run inside the TrustZone. For the evaluation, we used the TrustZone of a Raspberry Pi 3. We use the open-source trusted execution environment OP-TEE [43] as a secure-world operating system. The normal world runs a TrustZone-enabled Linux kernel.

As with Intel SGX (cf. Section 7.3), we again implement a trustlet providing a simple interface for receiving a pointer to a memory location. However, there are some subtle differences compared to the SGX enclave. First, trustlets are not allowed to simply access normal-world memory. To pass data or messages from normal world to secure world and vice versa, world shared memory is used, a region of non-secure memory, mapped both in the normal as well as in the secure world. With the world shared memory, we fulfill all criteria of Section 4.4.

On ARM, there are generally no unprivileged instructions to flush the cache or get a high-resolution timestamp [2]. However, they can be used from the operating system. Thus, in contrast to the double-fetch detection in syscalls or Intel SGX, we require root privileges to detect double fetches inside the TrustZone. This is not a real limitation, as we use the detection only for testing, and discovering bugs. An attacker using Flush+Reload as a trigger to exploit a double-fetch bug can rely on different time sources and eviction strategies as proposed by Lipp et al. [44].

Figure 10 shows a recorded cache trace of the trustlet. Similarly to Figure 11, a trace from Intel SGX, the cache hits are clearly distinguishable from the cache misses. Thus, we can detect double-fetch bugs in trustlets, even without having access to the corresponding binaries.

```
1  dropit_t config = dropit_init(1000);
2  dropit_start(config);
3  len = strlen(str); // First access
4  if (len < sizeof(buffer)) {
5    strcpy(buffer, str); // 2nd access,
6      // length of 'str' could have changed
7  } else {
8    printf("Too long!\n");
9  }
10 dropit_end(config, { printf("Fail!"); exit(-1);});
```

Listing 2: **Using DropIt to protect a simple string copy containing a double-fetch bug from being exploited. Only the highlighted lines (1, 2, and 10) have to be added to the existing code to eliminate the double-fetch bug.**

E EXAMPLE OF DROPIT

In this section, we show a small example of how to use DropIt. Listing 2 shows an example how to protect the insecure `strcpy` function using DropIt. A programmer only has to add 3 lines of code (highlighted in the listing) to protect arbitrary code from double fetch exploitation. DropIt is clearly simpler than current state-of-the-art software-based retry logic (cf. Algorithm 1).

DropIt is implemented in standard C without any dependencies on other libraries, and can thus be used in user space, kernel space, as well as in trusted execution environments (e.g., Intel SGX). If Intel TSX is not available, DropIt has the possibility to execute a fall-back function instead.