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

Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, Stefan Mangard

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 *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3196494.3196508

ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5576-6/18/06…$15.00
https://doi.org/10.1145/3196494.3196508

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

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. Pozniansky 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.

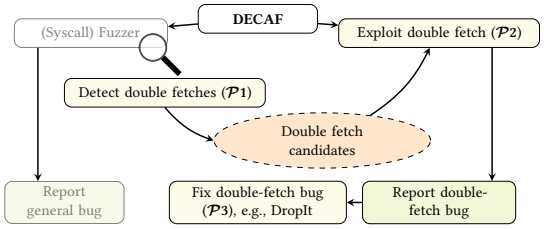

Figure 1: **An overview of the framework. Detecting (Primitive $\mathcal{P}1$) and exploiting (Primitive $\mathcal{P}2$) double fetches runs in parallel to the syscall fuzzer. Reported double-fetch bugs can be eliminated (Primitive $\mathcal{P}3$) after the fuzzing process.**

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`.

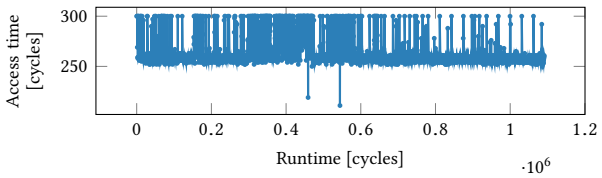

Figure 2: **Flush+Reload timing trace for a syscall with a double fetch. The two downward peaks show when the kernel accessed the argument.**

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.

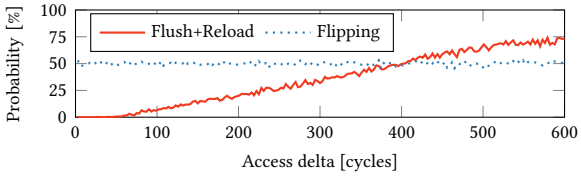

Figure 3: **The probability of successfully exploiting a double-fetch bug depending on the time between the accesses.**

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.

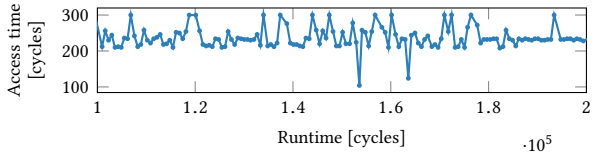

Figure 4: **The two memory accesses of the `FIDEDUPERANGE` `ioctl` in Linux kernel 4.5 to 4.7 can be clearly seen at around $1.5 \cdot 10^5$ and $1.6 \cdot 10^5$ cycles.**

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.

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.

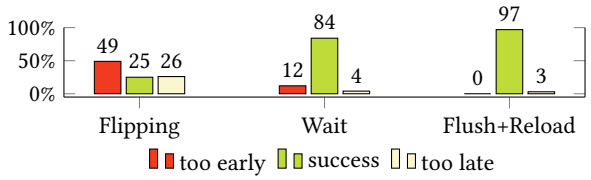

Figure 5: **Comparing three exploits for CVE-2016-6516. Our Flush+Reload-based trigger in TrinityDECAF succeeds in 97 %, outperforming the provided proof-of-concept (84 %) and the state-of-the-art method of value flipping (25 %).**

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

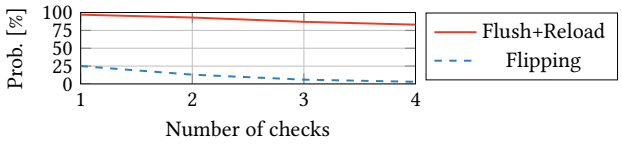

Figure 6: **The probability of double-fetch bug exploitation decreases with the number of sanity checks as it only succeeds if the value changes between the last two accesses.**

(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 ($\mathcal{P}$**2**) 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 ($\mathcal{P}$**2**) 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 $\mathcal{P}$1 on Trusted Execution Environments

We evaluate $\mathcal{P}$**1** 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 ($\mathcal{P}$**1**), 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 $\mathcal{P}$**1** on ARM TrustZone.

7.4 Evaluation of DropIt

To evaluate our open-source library DropIt, as an instantiation of $\mathcal{P}$**3**, 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.

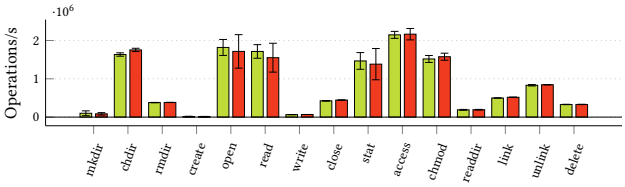

Figure 7: **The number of executed file operations per second of our re-implemented `getname_flags` using DropIt (green) does not significantly differ from the version in the vanilla kernel (red) measured with the IOZone Fileops benchmark.**

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.

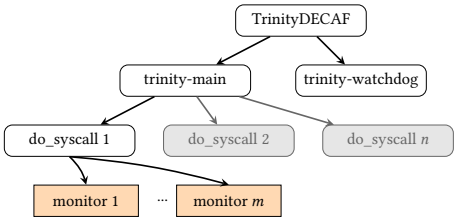

Figure 8: **The structure of TrinityDECAF with the Flush+Reload monitoring threads for the syscall parameters.**

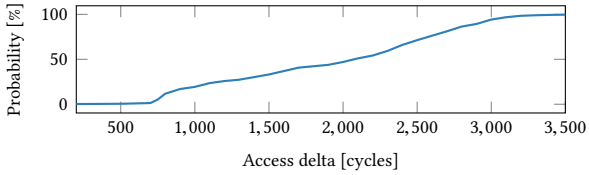

Figure 9: **The probability of detecting a double fetch depending on the time between the accesses.**

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.

```
1 // first access of dest_count
2 if (get_user(count,  &argp->dest_count)) { [...] }
3 // allocation based on dest_count
4 size = offsetof(struct file_dedupe_range __user,
5   info[count]);
6 same = memdup_user(argp, size);
7 if (IS_ERR(same)) { [...] }
8 ret = vfs_dedupe_file_range(file, same);
9 // 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

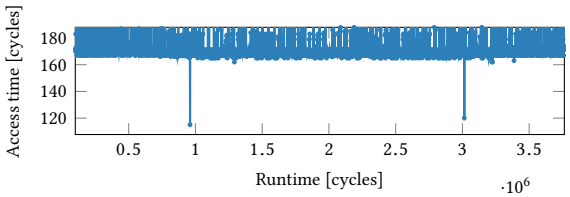

Figure 10: **A double fetch of a trustlet running inside the ARM TrustZone of a Raspberry Pi 3. The cache hits can be clearly seen at around $0.96 \cdot 10^6$ and $3.01 \cdot 10^6$ cycles as the access time drops from >160 cycles to <120 cycles.**

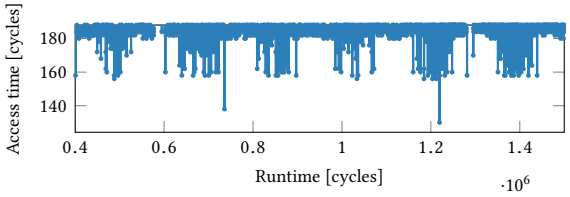

Figure 11: **Monitoring a double fetch inside an SGX enclave. The cache hits can be clearly seen at around $0.75 \cdot 10^6$ and $1.21 \cdot 10^6$ cycles as the access time drops from >150 cycles to <140 cycles.**

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.