Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution

Jo Van Bulck, *imec-DistriNet, KU Leuven;* Nico Weichbrodt and Rüdiger Kapitza, *IBR DS, TU Braunschweig;* Frank Piessens and Raoul Strackx, *imec-DistriNet, KU Leuven*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX

Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution

Jo Van Bulck
imec-DistriNet, KU Leuven
jo.vanbulck@cs.kuleuven.be

Nico Weichbrodt
IBR DS, TU Braunschweig
weichbr@ibr.cs.tu-bs.de

Rüdiger Kapitza
IBR DS, TU Braunschweig
kapitza@ibr.cs.tu-bs.de

Frank Piessens
imec-DistriNet, KU Leuven
frank.piessens@cs.kuleuven.be

Raoul Strackx
imec-DistriNet, KU Leuven
raoul.strackx@cs.kuleuven.be

Abstract

Protected module architectures, such as Intel SGX, enable strong trusted computing guarantees for hardware-enforced enclaves on top a potentially malicious operating system. However, such enclaved execution environments are known to be vulnerable to a powerful class of *controlled-channel* attacks. Recent research convincingly demonstrated that adversarial system software can extract sensitive data from enclaved applications by carefully revoking access rights on enclave pages, and recording the associated page faults. As a response, a number of state-of-the-art defense techniques has been proposed that suppress page faults during enclave execution.

This paper shows, however, that page table-based threats go beyond page faults. We demonstrate that an untrusted operating system can observe enclave page accesses *without* resorting to page faults, by exploiting other side-effects of the address translation process. We contribute two novel attack vectors that infer enclaved memory accesses from page table attributes, as well as from the caching behavior of unprotected page table memory. We demonstrate the effectiveness of our attacks by recovering EdDSA session keys with little to no noise from the popular Libgcrypt cryptographic software suite.

1 Introduction

Enclaved execution, or support for protected modules, is a promising new security paradigm that makes it possible to execute application code on a platform without having to trust the underlying operating system or hypervisor. With the advent of Intel SGX [32], support for Protected Module Architectures (PMAs) is now available on mainstream consumer hardware, and can be used to defend against malicious or compromised system software, both in an untrustworthy cloud environment [3, 36] as well as for desktop applications [18]. In particular, one line of research has developed techniques and supporting software to make it relatively easy to run unmodified legacy applications within an enclave [3, 2, 41, 45].

An essential aspect of enclaved execution is that the hardware prevents privileged system software from reading or writing a module's private memory directly, or from tampering with its internal control flow. However, the OS remains in charge of allocating platform resources (memory pages and CPU time) to protected modules, such that the platform can be protected against misbehaving or buggy enclaves. One consequence of this interaction between privileged system software and enclaves is an entirely new class of powerful, *indirect* attacks on enclaved applications. Xu et al. [48] first showed how a malicious OS can use page faults as a noise-free *controlled-channel* to extract rich information (full text and images) from a single run of a victim enclave. This is particularly dangerous when legacy software is running within an enclave, as these applications have not been hardened against side-channel attacks. As a result, several authors have expressed their concerns on side-channel vulnerabilities in a PMA setting in general, and the page fault channel in particular [12, 9, 43, 39, 7].

The research community has since proposed a number of compile-time and hardware-enabled defense techniques [40, 10, 39] that hide enclave page accesses from the OS. We argue, however, that page faults are but one side-effect of the address translation process that is observable by untrusted system software. More specifically, the main contribution of this paper is that we show that an adversarial OS can infer page accesses from an enclaved execution that *never* suffers a page fault. Our attacks exploit the key property that the SGX design leaves page table memory under explicit control of the untrusted OS. As such, other side-effects of the page table walk in enclave mode can be observed by the OS with very little to no noise. We identify and successfully exploit straightforward effects such as the setting of "accessed" and "dirty" bits, as well as less obvious effects such as the caching of page table memory itself. An important consequence is

that our novel attack vectors bypass recent defenses that focus exclusively on suppressing page faults [40, 39].

In summary, the contributions of this paper are:

- We advance the state-of-the-art by defeating recently proposed defense techniques, showing that we can infer page accesses without resorting to page faults.
- We present a page table-based technique to precisely interrupt an enclave at instruction-level granularity.
- We implement our novel attack vectors as an extension to Graphene-SGX's untrusted runtime, facilitating eavesdropping on unmodified applications.
- We demonstrate the effectiveness of our attacks by extracting private EdDSA session keys from the widely used Libgcrypt cryptographic library.

Our attack framework and evaluation scenarios are available as free software, licensed under GPLv3, at `https://github.com/jovanbulck/sgx-pte`.

2 Background

In this section, we provide the necessary background on Intel SGX, refine the attacker model, and discuss previous research results on controlled-channel attacks.

2.1 Intel SGX

Recent Intel x86 processors from Skylake onwards are being shipped with Software Guard eXtensions (SGX) [32, 1, 23] that enable strong, hardware-enforced trusted computing guarantees in an untrusted execution environment. SGX extends the instruction set and memory access logic of the Intel architecture to allow the execution of security-sensitive application logic in protected *enclaves* in isolation from the remainder of the system, including privileged OS or hypervisor.

Memory Protection. An SGX-enabled processor sets aside a contiguous physical memory area, referred to as Processor Reserved Memory (PRM). A hardware-level memory encryption engine guarantees the confidentiality, integrity, and freshness of PRM memory while it resides outside of the processor package. The PRM region is subdivided into two data structures: the Enclave Page Cache (EPC) and the Enclave Page Cache Map (EPCM). Protected 4 KB enclave code and data pages are allocated from the EPC, while every EPC page has a shadow entry in the EPCM to track ownership, type, address translation, and permission meta data. EPCM memory is exclusively managed by the processor, and is never directly accessible to software.

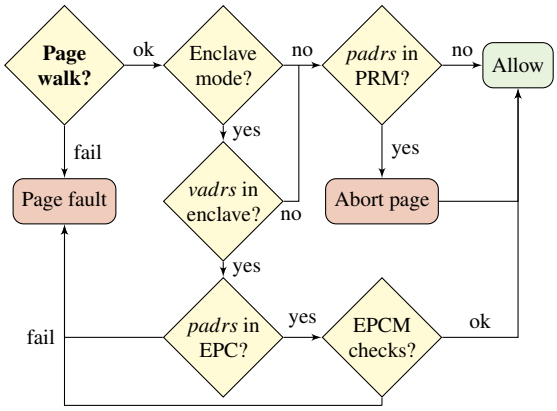

Figure 1: Additional memory access checks performed by SGX for a virtual address *vadrs* that maps to a physical address *padrs*.

SGX enclaves are instantiated as part of the virtual address space of a conventional OS process. Since PRM is a limited system resource, untrusted system software is in charge of assigning protected memory pages to enclaves, and is allowed to oversubscribe the EPC. At enclave creation time, the OS can instruct the processor to initialize newly allocated EPC pages with unprotected code or data. After finalizing the enclave, and before it can be entered, the hardware calculates a secure hash of the enclave's initial state. This allows the integrity of the untrusted loading process to be attested to a remote stakeholder [1]. SGX furthermore offers dedicated ring-zero instructions to securely evict and reload enclave pages between EPC memory and untrusted storage.

An important design decision of SGX is that it leaves page tables under explicit control of the untrusted operating system. Instead, SGX implements an additional, independent layer of access control on top of the legacy page table-based memory protection mechanism. Figure 1 summarizes the additional checks performed when accessing enclave memory. First, in order to translate the provided virtual address to a physical one, the processor traverses the OS-managed page tables, as well as the extended page tables set up by the hypervisor, if any. As usual, a page fault is signaled to the untrusted OS in case of a permission mismatch or missing page table entry during address translation. Any attempt to access the PRM region in non-enclave mode results in abort page semantics, i.e., read `0xFF` and ignore writes. Likewise, in enclave mode, the processor is allowed to reference all memory that falls outside of the executing enclave's virtual address range, but abort page semantics apply when such an address resolves into PRM memory. Furthermore, a page fault is signaled to the untrusted OS for EPC accesses that either do not belong to the currently executing enclave, are accessed through an unexpected virtual

address, or do not comply with the read/write/execute permissions imposed by the EPCM.

To speed up subsequent memory accesses, SGX employs the processor's Translation Lookaside Buffer (TLB) as a trusted cache of already checked page permissions. That is, SGX's memory access protection is entirely implemented in the Memory Management Unit (MMU) hardware that consults the untrusted page tables and the EPCM whenever a provided virtual address was not found in the TLB [32, 9]. SGX's security argument is based on the key observation that untrusted system software needs to interrupt the logical processor core before it can affect TLB entries. SGX therefore flushes the TLB and internal paging-structure caches whenever entering or exiting an enclave, and requires the OS to engage in a hardware-verified protocol that ensures proper TLB invalidation before evicting an EPC page.

SGX's dual permission lookup scheme prevents malicious system software from mounting active memory mapping attacks [9]. The output of the address translation process is considered untrusted, and the most restrictive of both permissions is applied. However, this design also implies that an attacker controlling page table permissions can cause enclave code to cause page faults, and be notified when certain pages are accessed. This property lies at the basis of the page fault attacks described in Section 2.3.

Enclave Entry and Exit. SGX enclaves are embedded in the address space of an untrusted user mode application, and can be internally multithreaded. They have to be explicitly entered by means of a dedicated `eenter` instruction that switches the logical processor to enclave mode, and transfers control to a predetermined entry point in the enclave's code section. The untrusted application context can exchange data with the enclave via unprotected memory. A processor running in enclave mode can be switched back programmatically by invoking the `eexit` instruction, or in case of a fault or external interrupt, through a process known as Asynchronous Enclave Exit (AEX). Upon AEX, the processor securely stores the execution context and exit reason (exception number) in a predetermined State Save Area (SSA) inside the enclave, and replaces CPU registers with a synthetic state before transferring control to the untrusted OS exception handler specified in the Interrupt Descriptor Table (IDT). In case of a page fault, SGX also takes care of zeroing out the twelve least significant bits of the faulting address, revealing only the page number, but not the 12-bit offset within that page.

Importantly, SGX enclave threads are unaware of interrupts by design, and have to be resumed explicitly by invoking `eresume` from the unprotected application context. The `eresume` instruction takes care of restoring the previously saved processor state, and redirects control flow to the instruction pointer specified in the SSA frame. SGX allows an enclave to register trusted in-enclave exception handlers with a cooperative OS. For this to work, `eenter` has to be explicitly called before `eresume`, so as to allow the previously interrupted enclave to inspect and modify its internal SSA frame. Since `eresume` cannot be intercepted however, an enclave has no way of enforcing its internal exception handler to be actually called.

SGX's exception model ensures that the untrusted operating system remains in control of shared platform resources such as memory or CPU time, and prevents direct information leakage of register contents. However, partial information on the enclave's internal state still leaks to the OS via exception vectors, and the access type and page base address in case of a page fault.

2.2 Attacker Model and Assumptions

The adversary's goal is to derive sensitive application data processed in an enclave. We assume the standard SGX threat model where an attacker has full control over privileged system software including the operating system and hypervisor. The attacker has full control over OS scheduling decisions; she can pin specific threads to specific CPU cores, and interrupt enclaves repeatedly. She can furthermore modify all non-enclaved parts of the application. Like previous SGX attacks [48, 40, 13, 37, 28], we finally assume knowledge of the (compiled) source code of the target application.

At the system level, we assume a classical MMU-based architecture where the system software maintains a multi-level page table data structure in OS memory to control virtual to physical page mappings. We assume the OS is in control of enclave page mappings, whereas the PMA guarantees the confidentiality and integrity of enclave pages, and properly verifies address translations to protect against page remapping attacks. Importantly, in contrast to previously published controlled-channel attacks discussed below, we assume a *PF-oblivious* attacker model where any page faults in enclave mode are hidden from untrusted system software. Our notion of stealthiness thus requires an attacker to infer page access patterns from an enclaved execution that *never* suffers a page fault. In addition, to stay under the radar of remote attestation schemes [39] that require the user's approval for each enclave invocation, our stealthy adversary should extract information from a *single* run of the victim enclave.

2.3 Controlled-Channel Attacks

This section briefly revisits previous research on page fault-driven attacks and defenses. We first explain how sensitive information can be derived from an enclave's

page fault behavior, and thereafter elaborate on recently proposed state-of-the-art defense techniques.

Tracking Page Faults. As explained above, a page fault during enclave execution triggers an AEX that hands over control to the untrusted operating system, revealing the base address of the faulting page. A malicious OS can exploit this property to obtain a page-level trace of enclave execution by clearing the "present" bit in the Page Table Entries (PTEs) that form the enclave's virtual address space. For maximal information leakage, an adversary allocates at most one code page and up to two operand data pages at all times. Furthermore, the access type can be inferred by manipulating the "writable" and "execute disable" PTE attributes.

Seminal work by Xu et al. [48] first showed how to exploit the page fault side-channel in a deterministic way. Their *controlled-channel* attacks exploit secret-dependent control flow and data accesses in unmodified legacy applications running on top of the SGX-based Haven [3] architecture. To overcome the coarse-grained page-level granularity, they observe that the *sequence* of preceding page faults can be used to uniquely identify a specific memory access. The controlled-channel attack relies on an exhaustive offline analysis of the target application binary to identify page fault sequences, and afterwards uses this information to extract rich information (full text and images) without noise from a single run of the victim enclave. Subsequent work by Shinde et al. [40] demonstrated that the page fault channel is sufficiently strong to extract cryptographic key bits from unmodified versions of OpenSSL and Libgcrypt.

Proposed Defenses. Ferraiuolo et al. [12] propose the use of dedicated CPU instructions to prevent certain pages from being swapped out of the protected memory area. This defense technique overlooks however that page faults can also be caused by directly modifying PTE attributes controlled by the OS. Shinde et al. [40] introduce the notion of *PF-obliviousness* which requires that any information leaked via page fault patterns can also be learned from running the program without inducing any page faults. They propose a compiler-based solution called *deterministic multiplexing* to generate PF-oblivious programs that unconditionally access all code and data pages at the same level of the execution tree. Without developer-assisted optimizations however, their approach exhibits unacceptable performance overheads [40] in practical application scenarios, which is why they also propose a hardware-assisted solution. In the *contractual execution* model, an enclave agrees with the untrusted OS that a number of sensitive pages remain mapped in its address space. The hardware is modified to report page faults directly to the enclave, without OS intervention, so as to enable protected enclave programs to detect contract violations. The enclave's fault handler can decide to either forward the page fault to the OS, abort the enclave program, or perform a fake execution to hide the page fault completely.

It seems that Intel made a first step towards supporting contractual execution on SGX platforms. As per revision 2 of the SGX specification [21], AEX can optionally store information about page faults in the interrupted enclave's SSA frame. This allows an SGX enclave to register a trusted exception handler for page faults. As explained in Section 2.1, however, the unprotected application can trivially `eresume` an enclave without first calling its designated exception handler. That is, the SGX v2 design still leaves enclaves explicitly unaware of interrupts or page faults. In response, Shih et al. [39] present a pragmatic approach to contractual execution on SGX platforms. Their solution, called T-SGX, leverages hardware support for Transactional Synchronization eXtensions (TSX) in recent Intel processors [23]. TSX was designed to synchronize the critical sections of multiple threads without the overhead of software-based locks. Code executing in a TSX transaction is aborted and automatically rolled back whenever encountering a cache conflict or exception. The security argument of T-SGX relies on the important property that a page fault during a TSX transaction immediately transfers control to a user-level transaction abort handler, without first notifying the OS. In case of an external interrupt on the contrary, the normal AEX procedure vectors to the OS, but TSX ensures that the in-enclave transaction abort handler is called on `eresume`. The T-SGX compiler wraps each basic block in a TSX transaction, and uses a carefully designed springboard page to hide page faults across transactions. Since TSX lacks hardware support to distinguish between page faults and regular interrupts in the abort handler, T-SGX restarts transactions by default, and only terminates the enclave program after counting too many consecutive aborts of the same transaction. Since the OS is made unaware of page faults, an adversary learns at most one page access by observing early program termination. T-SGX prevents reruns by requiring the remote enclave owner's consent before starting the enclaved application.

Note that T-SGX does *not* consider frequent enclave preemptions suspicious (up to 10 consecutive transaction aborts are allowed for each individual basic block). Between the submission and acceptance of this paper, however, more recent work was published [7] that leverages TSX to not only hide page faults, but also monitor suspicious interrupt rates. We discuss this heuristic defense technique and its implications for our attacks in more detail in Section 6.

Finally, Costan et al. [10] present a hardware-software co-design called Sanctum that represents a more radical

approach to eliminate controlled-channel attacks. Not only does Sanctum dispatch page faults directly to enclaves, but it also allows them to maintain their own virtual-to-physical mappings in a separate page table hierarchy in enclave-private memory. As further explored in Section 6, this design decision effectively prevents directed page table-based attacks from the OS. While Sanctum explicitly identifies information leakage from "accessed" and "dirty" page table attributes as a motivation for enclave-private page tables, we are the first to provide an exploitation strategy and to explore the implications of this side-channel.

3 Stealthy Page Table-Based Attacks

In this section, we present the design of our novel page table-based attacks. We first introduce two distinct ways in which a PF-oblivious attacker can detect page accesses after the enclave has programmatically been exited. Next, we present our approach to dealing with cached TLB entries for subsequent accesses to the same page. We finally explain how to infer conditional control or data flow in large programs by correlating subsequent page accesses in *page sets* as a more stealthy alternative to the page fault sequences introduced by Xu et al. [48].

3.1 Monitoring Page Table Entries

As a running example, consider the leftmost code snippet in Fig. 2, where we assume that *a* and *b* reference different data pages. In the classical controlled-channel attack [48, 40], an adversary would revoke access rights on both pages before entering the enclave, and learn the secret input by observing a page fault on either *a* or *b*.

```
1 void inc_secret (int s) {
2     if (s)
3         *a += 1;
4     else
5         *b += 1;
6 }
```

```
1 int compare_and_swap (
      int old, int new) {
2     if (*a == old)
3         return (*a=new);
4     else return *a;
5 }
```

Figure 2: Example code with secret-dependent data flow.

Our attacks are based on the important observation that a processor in enclave mode accesses unprotected page table memory during the address translation process. The key intuition is to exploit side-effects of the page table walk to identify which page has been accessed. In the following, we show that an adversary with access to unprotected page table memory can learn the secret input without resorting to page faults, either explicitly via page table attributes, or implicitly by observing cache misses.

A/D Bits. Since memory is a limited system resource, swapping out pages is benign OS behavior. To help memory-management software make an informed decision, Intel x86 processors [23] explicitly provide insight into an application's memory usage via page table attributes. The CPU's address translation logic sets a dedicated Accessed (*A*) bit whenever reading a page table entry, and takes care to set the Dirty (*D*) flag the first time a page has been written to. A/D attributes are stored in kernel-space memory, alongside the physical address of the page being referenced by the corresponding PTE entry, and need to be explicitly cleared by software.

We experimentally confirmed that A/D bits are also updated in enclave mode. An adversary inspecting these PTE attributes after enclave execution is thus provided with a perfect, noise-free information channel regarding the accessed memory pages. She can furthermore unambiguously distinguish between read and write accesses to the same page. In our `inc_secret` example, the secret input is directly revealed through the "accessed" bit of the PTEs referenced by *a* respectively *b*. The right-hand side of Fig. 2 provides a more subtle example where the data page referenced by *a* is first accessed, and thereafter either written to, or read again. An adversary can distinguish between these cases using the "dirty" PTE attribute. Note that a page fault-based attack could derive the same information using the "writable" attribute, if stealthiness is not a concern.

Cache Misses. Since modern CPUs can process data an order of magnitude faster than it can be fetched from DRAM, they rely on on an intricate cache hierarchy to speed up repeated code and data accesses. Contemporary Intel CPUs [23] feature three levels of multi-way, set-associative caches for instruction/data memory, and a separate TLB plus specialized paging-structure caches to accelerate address translation. Cache memories introduce a measurable timing difference for DRAM accesses and enable a powerful class of microarchitectural side-channel attacks, for they are shared among all software running on the platform.

A reliable and powerful class of access-driven cache attacks based on the FLUSH+RELOAD [50] technique exploits the availability of physical shared memory between the attacker and the victim, as is often the case with shared libraries. FLUSH+RELOAD relies on the `clflush` instruction that invalidates from the entire cache hierarchy all entries corresponding to a specified virtual address. To spy on a victim application, an adversary explicitly flushes a specified address in the shared memory region. Afterwards, she carefully times the amount of time it takes to reload the data, so as to determine whether or not the address has been accessed by the victim in the meantime.

One cannot directly apply FLUSH+RELOAD techniques

to SGX enclaves, since the `clflush` instruction requires read permissions on the provided memory location [23]. So it seems that properly implemented SGX enclaves do not share physical memory with their untrusted environment. We make the important observation, however, that an SGX enclave still *implicitly* shares unprotected page table memory with the operating system. Since page table entries are stored in regular DRAM, they are subject to the same caching mechanisms as any other memory location [23, 15] Additionally, modern Intel CPUs employ an internal paging-structure cache for page table entries that reference other paging structures (but not those that map pages), and cache physical addresses in the TLB. As explained in Section 2.1, the processor's internal TLB and paging-structure caches are cleared whenever entering or exiting an enclave. However, since the data cache hierarchy remains explicitly untouched, an adversarial OS can perform a FLUSH+RELOAD-based cache timing attack on the page table itself.

In our `inc_secret` running example, a kernel-space attacker uses `clflush` to evict the last-level PTEs referenced by *a* as well as *b*, before entering the enclave. After the enclave has returned, she learns the secret input by carefully recording the amount of time it takes to reload the relevant PTEs. The latter can easily be achieved on x86 processors using the `rdtsc` instruction. We experimentally ascertained a timing penalty of at least 150 cycles for PTE entries that miss the cache, practically turning our FLUSH+RELOAD page table attack into a reliable way to decide enclave page accesses.

Discussion. Cache timing attacks on page table memory reveal a fundamental flaw in the SGX design. That is, walking the untrusted page table during enclave execution discloses memory accesses at page-level granularity, even when faults would be suppressed and A/D bits are masked. However, as compared to the A/D channel, a cache-based attack suffers from a few limitations. First, one cannot distinguish between read and write accesses to the same page. This is not really a practical concern, however, since previous fault-based attacks [48, 40] do not rely specifically on write accesses. A second limitation considers the processor's prefetch unit [22, 17] that loads adjacent data speculatively into the cache. Specifically, during the reload phase of FLUSH+RELOAD, subsequent measurements might be destroyed. We develop a strategy to robustly infer page access patterns in the presence of false positives in Section 3.3.

A more severe limitation affects the granularity at which we can see page accesses. Since CPU caches exploit spatial locality, they fetch data from DRAM more than one byte at a time. The atomic unit of cache organization is called a *cache line* and measures 64 bytes on recent Intel processors [23]. A PTE entry on the other hand occupies only 8 bytes, implying that eight adjacent PTEs share the same cache line. PTE monitoring at a cache line granularity can thus conveniently be modelled as spying on enlarged ($8 * 4$ KB$= 32$ KB) pages.

```
1 point ec_mul (int d, point P) {
2   point Q = 0; int n = nbits(d);
3   for (int i = n−1; i >= 0; i−−) {
4     Q = point_double(Q);
5     if (d & (0x1 << i))
6       Q = point_add(Q, P);
7   }
8   return Q;
9 }
```

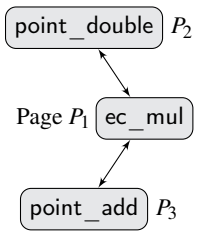

Figure 3: Elliptic curve scalar point multiplication.

3.2 Monitoring Repeated Accesses

So far, we only described how to detect memory page accesses after the enclave program has returned to its untrusted execution context. This suffices to extract secrets from the elementary code snippets in Fig. 2. More realistic scenarios, however, repeatedly operate over the same code or data in a single start-to-end run.

As an example, consider the pseudocode for elliptic curve scalar point multiplication in Fig. 3, where a provided point P is multiplied with a secret scalar d to obtain another point Q. The algorithm uses the double-and-add method, a variation of square-and-multiply used for modular exponentiation in a.o. RSA, and widely studied in side-channel analysis research [26, 8, 49, 50, 40]. We elaborate more on elliptic curve cryptography, and successfully attack Libgcrypt's implementation of the algorithm in Section 5.2. For now, we assume the `ec_mul` function is situated on code page P_1, whereas the subroutines `point_double` and `point_add` are located on distinct pages P_2 and P_3. Previous fault-driven attacks [40] recovers the private scalar by observing different page fault sequences for iterations corresponding to a one $(P_1, P_2, P_1, P_3, P_1, P_2)$ or zero (P_1, P_2, P_1, P_2) bit.

The key difference in our stealthy attacker model, as compared to the page fault channel, is that we are *not* notified in case of a memory access. Instead, page table entries should be explicitly monitored to establish whether they have been accessed or not. If the adversary only probes PTEs after enclave execution, she is left with aggregated information only (e.g., all pages P_1, P_2, and P_3 have been accessed). We therefore introduce a dedicated spy thread that monitors PTE entries in real-time, while the victim executes. The main challenge now becomes that SGX caches address translations in the TLB, implying that only the first access to a specific page results in a

page table walk. Subsequent accesses to the same page most likely hit the TLB, and will not be observed by a spy thread monitoring page table memory. In the following, we present our approach to overcoming this challenge.

Flushing the TLB. We explicitly interrupt the enclaved victim application in order to reliably evict cached address translations without provoking page faults. Note that we don't even have to invalidate TLB entries explicitly, since an SGX-enabled processor automatically takes care of this during the AEX process. An adversary is left with two choices. She can either periodically interrupt the enclave with a timer-based preemption, or she can conditionally interrupt the victim CPU from a snooping thread. The timer-based approach would have to interrupt the victim enclave at a high frequency to minimize the risk of missing page accesses. Since SGX leaves enclaves interrupt-unaware by design, they have no way of detecting these frequent preemptions. Some of the enhanced PMA designs [10, 39] targeted by our stealthy attacker, however, redirect interrupts as well as page faults to a trusted enclave entry stub. Such fortified enclaves could recognize suspicious interrupt rates as an artefact of the attack, defeating our argument for stealthiness. We therefore opted for the second option that conditionally interrupts the victim CPU minimally. In this respect, note that concurrent, unpublished work [46] has demonstrated that Intel's HyperThreading technology can be abused to evict TLB entries from a co-resident logical processor in real-time, *without* interrupting the victim enclave.

Our spy thread monitors one or more page table entries in a tight loop, preempting the victim enclave CPU after a page access has been detected. The latter can be easily achieved in multiprocessor systems through a directed Inter-Processor Interrupt (IPI), specifically designed to a.o., synchronize address translations across cores. From the point of view of the enclave, IPIs are directly handled by the CPU's local Advanced Programmable Interrupt Controller (APIC), and are thus indistinguishable from regular interrupts sent by a benign operating system.

Monitoring A/D Bits. We experimentally confirmed that the "accessed" PTE attribute is only updated during the first page walk, since subsequent accesses hit the TLB. Furthermore, we found that the "dirty" attribute is independently set once for the first subsequent write access to that page. In the A/D implementation of our spy thread, an IPI is sent as soon as the *A* bit of the monitored PTE entry flips. Alternatively, an adversary can choose to only interrupt the victim enclave when the *D* flag changes. This might allow for a slightly stealthier attack, which interrupts the victim minimally, as pages are typically more often read than written to.

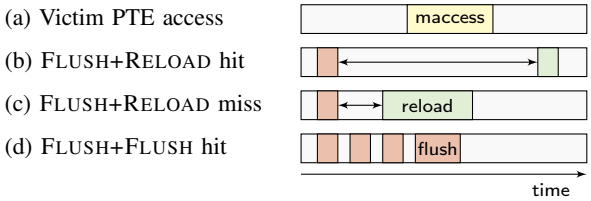

Figure 4: FLUSH+FLUSH as a high-resolution, low-latency channel to spy on victim PTE memory accesses.

Monitoring PTE Memory Accesses. In a classical FLUSH+RELOAD attack [50], time is divided into slots. The spy program flushes the monitored cache lines at the start of each time slot, and reloads them at the end to find out whether they have been accessed by the concurrent victim program executing independently. When the victim's memory access overlaps with the flush or reload phases of the spy thread however, the measurement might be lost, as illustrated in Fig. 4c. Naturally, the probability of an overlapping victim access increases as the length of the time slot decreases, whereas a longer time slot increases detection latency and might miss subsequent memory accesses by the victim. As such, a trade-off is presented between attack resolution and accuracy.

When reloading PTEs after the enclave has been exited, as in the start-to-end examples of Fig. 2, our measurement cannot be destroyed by a concurrent victim access. This is not the case, however, when monitoring page table memory in real-time from a spy thread. Moreover, the victim only makes a single memory access to the monitored PTE entry, for subsequent accesses to the same page hit the TLB. In a classical FLUSH+RELOAD attack on the other hand, a missed memory access can be compensated for by subsequent accesses in the next time slot. We therefore chose to adopt a novel technique called FLUSH+FLUSH [16] that abuses microarchitectural timing differences in the execution time of the x86 `clflush` instruction, which depends on whether the data is cached or not. A spy thread that repeatedly flushes a specific PTE entry will observe a slightly higher execution time when the page has been accessed by the victim, as illustrated in Fig. 4d. Spying on page table memory the FLUSH+FLUSH way thus ensures we can see all page accesses with a minimal detection latency.

FLUSH+FLUSH also confronts us with a new challenge however, since the microarchitectural timing differences of the `clflush` instruction are inherently more subtle than the apparent timing penalty for a DRAM access in FLUSH+RELOAD [16]. On the bright side, `clflush` does not trigger the processor's prefetcher, and therefore does not destroy subsequent measurements, a known concern for FLUSH+RELOAD [17]. We furthermore remark that, if needed, the spy thread can be made more robust by

monitoring multiple code or data PTEs that each should be accessed before sending the IPI.

3.3 Inferring Page Access Patterns

An essential ingredient of the attack procedure outlined so far, is that we interrupt the victim enclave via a targeted IPI from the spy thread. Some time passes however before the victim is interrupted, since the spy CPU cannot instantaneously detect PTE accesses and send the IPI. During this time interval, the victim enclave continues to execute instructions that may access additional code and data pages. Previous controlled-channel attacks on the contrary instantaneously trap to the OS in case of a page fault. This enables a PF-aware adversary to unambiguously distinguish two successive enclave instructions, whereas the accuracy at which we can see subsequent page accesses is constrained by IPI latency. In this respect, a fault-driven attack can be modelled as having zero latency between detecting a page access and interrupting the victim.

Page Fault Sequences. Naturally, page table-based attacks have to deal with the limitation that they can only see memory accesses at a page-level granularity. Since functions as well as data objects typically share the same memory page with other functions or data objects, one cannot directly identify specific function or data accesses in a large enclave program. Xu et al. [48] overcome this challenge by identifying unique *page fault sequences* that lead to a particular code or data access. Since a PF-aware attacker does not have to cope with latency in the measurement process, she may construct page access sequences at instruction-level granularity.

In the running example of Fig. 3, the `ec_mul` function on P_1 serves as a trampoline to redirect control flow to either `point_double` on page P_2 or `point_add` on page P_3, based on the secret scalar bit under consideration. A one bit can be identified by the sequence $(P_2, P_1, P_3, P_1, P_2)$. An observed page fault sequence of (P_2, P_1, P_2) on the other hand, corresponds to an iteration with a zero bit. One approach would be to implement a state machine in the spy thread to recognize such sequences. However, as the intermediate P_1 accesses are only a few instructions long, they could be easily missed by a stealthy spy that has to take IPI latency into account. Moreover, page fault sequences presuppose a completely noise-free way of establishing enclave page accesses. Recall from the above discussion, however, that FLUSH+RELOAD may suffer from occasional false positives by triggering the processor's prefetcher.

Page Sets. To correlate subsequent page accesses in large enclave programs, we introduce the notion of *page sets* as a robust alternative to page fault sequences. Our spy thread continuously monitors one or more PTEs, from here on referred to as the *trigger page(s)*, and interrupts the victim enclave as soon as an access is detected. Upon IPI arrival, the spy establishes the set of pages (not) accessed by the victim, using one of the techniques from Section 3.1. Since the TLB is cleared whenever entering or exiting the enclave, these pages must have been accessed at least once by the victim from the previous interrupt up to now. We make the key observation that specific points in the execution trace of a large enclave program can be uniquely identified by matching the pattern of all pages accessed or not accessed in between two successive accesses to a trigger page. Note that information recovery via page sets is inherently stealthier than the previously proposed page fault sequences [48, 40] in that victim enclaves are only interrupted when accessing the trigger page. Where a page fault only leaks one bit of information (i.e., the trigger page was accessed), our notion of page sets allows a spy to capture the maximum information for every trigger page interrupt.

Applying our page set theory to the running example of Fig. 3, the spy thread monitors the trigger page P_2 holding a.o., `point_double`, and matches the page set $\{P_1, P_3\}$ on every interrupt. If both P_1 (`ec_mul`) and P_3 (`point_add`) have been accessed, the iteration corresponds to a one bit. Likewise, if P_1 has been accessed, but not P_3, the iteration processed a zero bit. Finally, in case P_1 as well as P_3 were both not accessed, P_2 must have been accessed from an execution context other than the targeted `point_double` invocation, and we classify the interrupt as a false positive.

After identifying secret-dependent control flow or data accesses in the victim application, a successful attack comes down to designating specific pages to be tracked in the spy thread, and recognizing the associated page set patterns. Analogous to previous fault-based attacks [48, 40], we first perform a detailed offline analysis of the enclaved application binary to extract an ideal trace of instruction-granular page accesses for a known input. From this ideal trace, we select a suitable candidate trigger page, and we construct the sets of all pages accessed or not accessed in between two hits on the trigger page. By comparing the resulting page sets, we are left with a page set pattern that (uniquely and robustly) identifies a specific point in the victim's execution trace.

4 Implementation

Similar to previous controlled-channel attacks [48, 40], our exploits target unmodified legacy applications running under the protection of a PMA. The enclaved application binary is protected from the untrusted host operating system by means of a *shielding system* that provides trusted library services, and interposes on system calls. Previ-

ous controlled-channel attacks on Intel SGX were implemented for the Haven [3] shielding system. Since Haven is not publicly available, we implemented our attacks on the open-source[1] Graphene-SGX library OS [45]. We first briefly overview the internals of Graphene-SGX, and thereafter explain how we extended the untrusted runtime with a reusable attacker framework.

Graphene-SGX. Library OSs such as Graphene [44] repackage conventional OS kernel services into a user-mode application library. System calls made by the legacy application are transparently transformed into libOS function calls, which are then either processed locally, or translated into a minimal host kernel ABI that provides core OS primitives. The libOS relies on a small Platform Adaptation Layer (PAL) to translate platform-independent host ABI calls into a narrow set of system calls to the underlying host operating system, which remains, however, explicitly trusted from a security perspective.

Graphene-SGX [45] – like other recently proposed SGX-based shielding systems including Haven [3], Panoply [41], and SCONE [2] – improves over this situation by not only protecting libOS instances from each other, but also from a malicious host operating system. To this end, Graphene-SGX encapsulates the entire libOS, including the unmodified application binary and supporting libraries, inside an SGX enclave. Graphene also inserts a trusted runtime with a customized C library and ELF loader in the enclave. Since SGX prohibits enclaves from making system calls directly, the PAL is split into a trusted part that calls out to an untrusted runtime in the containing application to perform the system call to the untrusted host OS. Graphene-SGX furthermore relies on an untrusted Linux driver for enclave creation/tear down and protected memory management via the dedicated ring-zero SGX instruction set.

Attack Framework. We implemented our attacks as an extension to Graphene's untrusted runtime, leaving the trusted in-enclave components unchanged. Our implementation is conceived as a reusable framework to facilitate eavesdropping on different application binaries.

Figure 5 summarizes the steps undertaken by our attack framework. ① The untrusted user space runtime creates a separate spy thread just before entering the enclave's main function. We affinitize the spy and victim threads to their own physical CPU cores to avoid any noise from page table shoot downs by the OS scheduler. ② The newly created spy thread continues its execution in kernel space by calling to our modified Graphene-SGX driver. We run our core attacker code in kernel mode to be able to easily send IPIs, inspect PTE attributes, and monitor page table memory. ③ The spy first goes through a pluggable, attack-specific initialization phase that creates the page sets to be monitored. ④ After synchronizing with the victim thread, which is still waiting to enter the enclave, the spy enters a tight probing loop that measures either `clflush` execution time, or A/D attributes of one or more page table entries. ⑤ Victim thread enters the enclave. ⑥ Upon detecting an access on the trigger page, the spy interrupts the victim thread as soon as possible. ⑦ The IPI handler on the victim CPU now establishes the access pattern for the monitored page set using either the noise-free FLUSH+RELOAD or A/D mechanism. Page set access patterns are logged for later parsing by an attack-specific post-processing script. ⑧ Spy and victim threads synchronize once more before resuming the enclave.

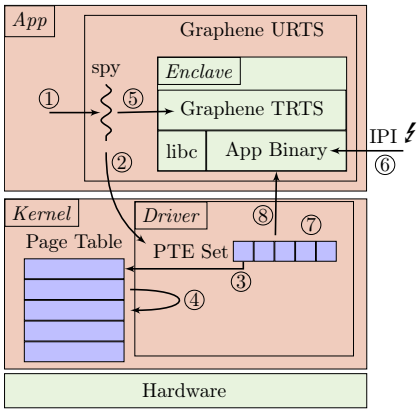

Figure 5: Graphene-SGX attack framework interaction.

So far, we assumed the attacker obtained the page addresses to be monitored from an `objdump` of the application binary. Graphene, like other SGX-based shielding systems [3, 41, 2], does not randomize the base address of loaded executables. Instead, applications and supporting libraries (including libc) are loaded at deterministic memory locations. To easily discover executable base addresses, we propose to first deploy the target application binary in an attacker-controlled libOS instance that we minimally modified to leak load addresses. SGX's remote attestation scheme properly prevents us from deploying the modified libOS instance when running the application for the remote stakeholder, but the observed load addresses will be identical. Note that it has been shown [48] that hypothetical support for conventional address space layout randomization, which only randomizes the application's base address, could be easily defeated by observing page access patterns.

Inter-Processor Interrupts. In a page fault-driven attack, the victim enclave is exited immediately when accessing a monitored page. For our PF-oblivious attacks

[1] `https://github.com/oscarlab/graphene`

on the contrary, we define IPI latency as the number of instructions executed by the victim enclave after accessing a trigger page, and before being interrupted by the spy thread. Reducing IPI latency is an important implementation consideration in that it defines the accuracy at which we can see subsequent page accesses. Before quantifying latency in the evaluation section, we present some general implementation techniques to minimize IPI latency.

Our driver hooks into an unused IPI vector of Linux's KVM subsystem by registering the address of our interrupt handler in the system-wide IDT. This allows us to send the IPI promptly from assembly code in the spy thread by writing to the relevant memory-mapped APIC address, instead of having to rely on Linux's IPI subsystem that performs bookkeeping on shared data structures before sending the interrupt. To further reduce IPI latency, we considered a previously proposed [28] technique that sets the "cache disable" bit in the `CR0` control register to disable the L1, L2, and L3 cache on the CPU running the victim enclave. We experimentally confirmed that this technique dramatically slows down the victim thread, and substantially reduces the number of instructions executed after accessing a trigger page. However, setting `CR0.CD` on the victim CPU invalidates our cache-based PTE timing attack vector. Moreover, the aforementioned T-SGX defense [39] would be able to detect this technique, for TSX relies on the CPU cache to start transactions [23].

Analyzing Page Sets. With our attack framework in place, the main challenge left is to select the pages that need to be tracked in the spy thread. To study the behavior of target applications, previous controlled-channel attacks [48] record a complete, byte-granular trace of page fault addresses by running the application outside of the enclave with at most one code and data page allocated at all times. We simplify this process via a GNU debugger script that extracts an instruction-granular code page trace by single-stepping through the *unprotected* application binary, recording the symbolic name and virtual page address of the instruction pointer. Furthermore, by placing strategic breakpoints, the debugger script can easily be instrumented to mark individual loop iterations.

To construct the most stealthy attack, we select a trigger page that is minimally accessed in the extracted trace, and we compose a set of remaining pages that unambiguously identifies the code page access of interest. When running the attack on an enclaved application binary, our driver dumps page set patterns for all accesses on the trigger page. Afterwards, we use a small, attack-specific post-processing script to match the desired patterns in the driver output. If needed, the pattern to be matched, can also include the page sets of previous or succeeding trigger page accesses, and can be made more robust by means of a regular expression.

Table 1: IPI latency in terms of the number of instructions executed by the victim after accessing the trigger page.

	ACCESSED		FLUSH+FLUSH		
Experiment	**Mean**	σ	**Mean**	σ	**Zero %**
`nop`	431.70	34.11	0.65	17.65	99.84
`add` register	176.30	14.60	0.15	6.18	99.94
`add` memory	32.45	2.79	0.06	1.92	99.88
`nop` nocache	0.02	0.39	–	–	–

5 Evaluation

In this section, we evaluate our attack framework. We first provide microbenchmarks to quantify IPI latency, and thereafter demonstrate the effectiveness of our attacks by extracting EdDSA session keys from an unmodified binary of the widely used Libgcrypt cryptographic library.

All experiments were conducted on publicly available off-the-shelf SGX hardware. We used a commodity Dell Inspiron 13 7359 laptop with a Skylake dual-core Intel i7-6500U processor and 8 GB of RAM. The machine runs Ubuntu 15.10, with a generic 64-bit Linux 4.2.0 kernel. To prevent any noise from OS scheduling decisions, we disabled HyperThreading and reserved a dedicated CPU for the spy thread using Linux's `isolcpus` boot option. We based our attack framework on a recent `master` checkout of the Graphene project, compiled with `gcc` v5.2.1.

5.1 IPI Latency Microbenchmarks

Recall from Section 4 that we want to minimize the number of instructions executed by the victim enclave after accessing a trigger page, and before being interrupted by a targeted IPI from the spy thread. In order to reliably quantify IPI latency, we wrote a small microbenchmark application that first accesses an isolated memory page, and immediately thereafter starts executing an instruction slide of 5,000 identical x86 instructions. For the microbenchmark experiments, we instrumented our driver to retrieve the instruction pointer stored in the SSA frame of the interrupted debug enclave through the `edbgrd` SGX instruction. The exact number of instructions executed in the microbenchmark application can be inferred by comparing the retrieved instruction pointer with the known start address of the instruction slide.

Interrupt Granularity. Table 1 records IPI latencies for different x86 instructions. We repeat all experiments 10,000 times for a spy thread that monitors the trigger page through the "accessed" PTE attribute, as well as for a spy that repeatedly flushes page table memory locations. We present the mean and the standard deviation (σ) to characterize IPI latency distributions. In the first

experiment, we prepare an instruction slide with ordinary no-operations. The upper row of Table 1 reveals a first important result. That is, our benchmark enclave can only be interrupted by an A/D spy at a relatively coarse-grained granularity of about 430 `nop`s, whereas the novel FLUSH+FLUSH technique immediately interrupts the victim thread. Note that interrupts with zero IPI latency arrive *within* the instruction that accessed the trigger page, even before the next enclave instruction started executing. The last column, which lists the percentage of interrupts with zero IPI latency, distinctly shows that a victim thread monitored by a FLUSH+FLUSH spy is interrupted within the trigger instruction with very high probability (99.84%). As such, FLUSH+FLUSH represents a *precise*, instruction-granular, technique to interrupt victim enclaves, improving significantly over related state-of-the-art enclave execution control proposals [47, 28, 33]. We furthermore found the technique to be *reliable*, for FLUSH+FLUSH recorded all 10,000 page accesses, without false positives, and with significantly less noise (smaller standard deviation) than an A/D spy.

The increased advantage of a FLUSH+FLUSH spy, as opposed to a spy monitoring A/D bits, can be understood from the effects on the caching behavior of the page table walk. A PTE memory location that is continuously probed by an A/D spy will be cached when the victim CPU performs the page table walk, whereas a FLUSH+FLUSH spy actively ensures the victim CPU misses the cache. As such, instructions that access the trigger page will take longer to complete, providing a wider time frame for IPI arrival. This effect is further aggravated when the processor needs to update the "accessed" page table attribute. For the victim CPU needs to perform another memory access to reload the PTE entry from DRAM when the *A* bit was not set, and the corresponding cache line has been flushed by a concurrent spy thread. Interestingly, we found that the victim's second PTE memory access, where the *A* bit is updated, is more noticeable from a FLUSH+FLUSH spy thread. Intel's software optimization manual [22] indeed confirms that "flushing cache lines in modified state are more costly than flushing cache lines in non-modified states".

Instruction Latency. The second and third experiments investigate the influence of the microbenchmark instruction type on IPI latency. We start from the intuition that an individual `nop` instruction is trivial to execute and can easily be pipelined, allowing many instructions to be executed in the limited time period after accessing the trigger page and before IPI arrival. The second row of Table 1 confirms that a victim program can make significantly less progress on an instruction slide with `add` instructions that sequentially increment a processor register. Likewise, the third row shows that IPI latency drops even further when the victim executes a sequence of `add` instructions that increment a memory location. The latter can be explained from the additional page table walk that retrieves the physical memory address of the data operand for the first `add` instruction.

Finally, we performed an experiment that entirely disables instruction and data caching on the victim CPU by setting the `CR0.CD` bit, as explained in Section 4. The last row of Table 1 clearly shows that this approach can almost completely eliminate IPI latency (mean and standard deviation near zero) for an A/D spy. This confirms our hypothesis that the observed IPI latency differences stem from the caching behavior of the page table walk. Of course, a FLUSH+FLUSH spy cannot see page accesses when the cache is disabled on the victim CPU.

5.2 Attacking Libgcrypt EdDSA

To illustrate the applicability of our attacks on real-world applications, we extract private EdDSA session keys from a general purpose cryptographic library Libgcrypt, which used in a.o., the popular GnuPG cryptographic software suite. More specifically, we reproduce a previously published [40] page fault-driven attack on Libgcrypt, showing that our stealthy attack vectors can extract the same information without triggering any page faults. Since Libgcrypt is officially distributed from source code, we built unmodified binaries for Libgcrypt v1.6.3 and v1.7.5 as well as the accompanying error-reporting library Libgpg-error v1.26 through the default `./configure && make` invocation, using `gcc` v5.2.1.

EdDSA Implementation. The Edwards-curve Digital Signature Algorithm (EdDSA) [4] is an efficient, high-security signature scheme over a twisted Edwards elliptic curve with public reference point G. The security of elliptic curve public key crypto systems critically relies on the computational intractability of the elliptic curve discrete logarithm problem: given an elliptic curve with two points A and B, find a scalar k such that $A = kB$. Recall that our running example in Fig. 3 provides an efficient algorithm for the inverse operation, i.e., multiply a point with a known scalar. EdDSA uses scalar point multiplication for public key generation, as well as in the signing operation. The private key d is derived from a randomly chosen large scalar value, and the corresponding public key is calculated as $Q = dG$. To sign a message M, EdDSA first generates a secret *session key* r, also referred to as *nonce*, by hashing the long-term private key d together with M. Next, the signature is calculated as the tuple $(R = rG, S = r + hash(R,Q,M)d)$. It can be seen that an adversary who learns the secret session key r from side-channel observation during the signing process, can easily recover the long-term private key as

```
1  if (mpi_is_secure (scalar)) {
2    /* If SCALAR is in secure memory we assume that it is the
3       secret key we use constant time operation.  */
4    point_init (&tmppnt);
5
6    for (j=nbits-1; j >= 0; j--) {
7      _gcry_mpi_ec_dup_point (result, result, ctx);
8      _gcry_mpi_ec_add_points (&tmppnt, result, point, ctx);
9      if (mpi_test_bit (scalar, j)) /* ← eliminated in v1.7.5 */
10       point_set (result, &tmppnt);
11   }
12   point_free (&tmppnt);
13 } else {
14   for (j=nbits-1; j >= 0; j--) {
15     _gcry_mpi_ec_dup_point (result, result, ctx);
16     if (mpi_test_bit (scalar, j))
17       _gcry_mpi_ec_add_points (result, result, point, ctx);
18   }
19 }
```

Figure 6: Scalar point multiplication in Libgcrypt v1.6.3.

$d = (S-r)/hash(R,Q,M)$, with (R,S) a valid signature for a known message M [4, 49].

Figure 6 provides the relevant section of the scalar point multiplication routine in Libgcrypt v1.6.3. Lines 14 to 18 are a straightforward implementation of Fig. 3, and have previously been successfully targeted in a page fault-aware attacker model [40]. We remark however that Libgcrypt provides some protection against side-channel attacks by tagging sensitive data, including the EdDSA long-term private key, as "secure memory" [25]. Lines 1 to 12 show how a hardened, add-always scalar point multiplication algorithm is applied when the provided scalar is tagged as secure memory. However, while the hardened algorithm of Libgcrypt v1.6.3 greatly reduces the attack surface by cutting down the amount of secret-dependent code, we show that even the short if branch on line 9 remains vulnerable to page table side-channel attacks during the public key generation phase. We verified that this defect has been addressed in the latest version v1.7.5 by replacing the if branch with a truly constant time swap operation. We also found, however, that Libgcrypt v1.6.3 as well as v1.7.5 do *not* tag the secret EdDSA session key as secure memory, resulting in the non-hardened path being taken during the signing phase.[2]

Monitoring A/D Bits. We first explain how we attacked the hardened multiplication (lines 6 to 11) in Libgcrypt v1.6.3. We found that every loop iteration accesses 21 distinct code pages, regardless of whether a one or a zero bit was processed. Our stealthy spy thread monitors the *A* attribute of the trigger page table entry holding the physical page address of `point_set`, which is accessed 126 or 127 times each iteration, depending on the scalar bit under consideration. We rely on a robust PTE set of nine additional code pages whose combined *A* bits unambiguously identify an unconditional execution point in `add_points` as well as the conditional `point_set` invocation on line 10. We refer the interested reader to Appendix A for the complete page sets of the Libgcrypt attacks. Our post-processing script reliably recovers the full 512-bit EdDSA session key by counting the number of IPIs (i.e., trigger page accesses) in between two page set pattern hits. PTE set hits are classified as belonging to a different iteration when the number of IPIs in between them exceeds a certain threshold value. As such, iterations that processed a one bit are easily recognized by two page set hits, whereas zero iterations hit only once. Our A/D attack on Libgcrypt v1.6.3 interrupts the victim enclave about 60,000 times.

To attack the standard multiplication (lines 14 to 18) in the latest Libgcrypt v1.7.5, we spy on the *A* attribute of the PTE that references the `test_bit` code page. Our offline analysis shows that the trigger page is accessed 93 or 237 times for iterations that respectively process a zero or a one bit. The spy thread records a PTE set of four additional code pages whose combined access patterns uniquely identify the if branch on line 16. We reliably recover all 512 secret scalar bits at post-processing time by observing that the PTE set pattern repeats exactly once every loop iteration, and the page set value for the first subsequent trigger page access depends on whether the if branch was taken or not. We counted only about 40,000 IPIs for our A/D attack on Libgcrypt v1.7.5.

Monitoring Cache Misses. Recall from Section 3 that spying on page table memory at a cache line granularity is challenging in that we can only see accesses for conceptually enlarged 32 KB pages. Our offline analysis on Libgcrypt v1.7.5 shows that every loop iteration accesses 22 code pages, belonging to three different application libraries: Libgcrypt, Libgpg-error, and the trusted libc included by Graphene. Only 11 of these 22 code pages fall in distinct cache lines. Interestingly, we found that the `free` wrapper function used by Libgcrypt stores/restores the `errno` memory location of the trusted in-enclave libc 46 or 102 times for zero respectively one iterations. The address of the error number for the current thread can be retrieved via the `__errno_location` function, residing at a remote location within the libc memory layout.

Our stealthy FLUSH+FLUSH spy uses the code page for the `__errno_location` libc function as a reliable trigger page that does not share a cache line with any of the other pages accessed in the loop. Our cache-based attack on Libgcrypt interrupts the victim enclave about 130,000 times for a single, start-to-end run. We furthermore construct a page set covering 7 distinct PTE cache lines that are recorded by the spy on every trigger page access, using the FLUSH+RELOAD technique after interrupting the

[2] To address this shortcoming, we contributed a patch that has been merged in Libgcrypt v1.7.7.

enclave. While the extracted page set value sequences themselves appear quite noisy at first sight, we found that certain values unmistakably repeat more often in iterations that processed a one bit. Furthermore, the number of IPIs (i.e., `errno` accesses) in between these values exhibit clear repetitions. Our post-processing script uses a regular expression to identify a robust pattern that repeats once every iteration. Again, key bits can be inferred straightforwardly from the number of IPIs in between pattern hits. Using this technique, we were able to correctly recover 485 bits of a 512-bit secret EdDSA session key in a single run of the victim enclave. Moreover, using the number of IPIs in between two recovered scalar bits as a heuristic measure, our post-processing script is able to give an indication of which bit positions are missing.

6 Discussion and Mitigations

Frequent Enclave Preemption. Our work shows that enclave memory accesses can be learned by spying on unprotected page tables, without triggering any page faults. This observation is paramount for the development of defenses against page table-based threats. Specifically, state-of-the-art PF-oblivious defenses [40, 39] do not achieve the required guarantees. We only interrupt the enclave when successive accesses to the same page need to be monitored. Importantly, our attacks remain undetected by T-SGX [39], since it allows up to 10 consecutive transaction aborts (interrupts) for each individual basic block. We do acknowledge, however, that the number of interrupts reported for our Libgcrypt attacks in Section 5.2 is substantially higher than what is to be expected under benign circumstances. We can therefore see improved, heuristic defenses using suspicious interrupt rates as an artefact of an ongoing attack.

Indeed, Déjà Vu [7], which was first published after we submitted this work, explores the use of TSX to construct an in-enclave reference clock thread that cannot be silently stopped by the OS. The enclave program is instrumented to time its own activity, so as to detect the execution slowdown associated with an unusual high number of AEXs. While Déjà Vu would likely recognize frequent enclave preemptions as a side-effect of our current attack framework, we argue that heuristic defenses do not address the *root* causes of page table-based information leakage. That is, our novel attack vectors are still applicable, and depending on the victim program, interrupts may not even be required. The knowledge that a specific page is accessed, can reveal security-sensitive information directly, or enable an attacker to launch a second phase of her attack [47]. Furthermore, as part of the continuous attacker-defender race, we expect the contributed attack vectors to trigger improved, stealthier attacks that remain under the radar of Déjà Vu-like defenses.

In this regard, during the preparation of the camera-ready version of this paper, we became aware of concurrent, non-peer-reviewed research [46] that independently developed page table-based attacks similar to ours. In contrast, their work focusses on the A/D channel rather than PTE caching, and shows that HyperThreading technology allows TLB entries to be evicted *without* interrupting the victim enclave. As such, they effectively demonstrate that Déjà Vu-like defenses are inherently insufficient to eliminate page table-based threats.

Hiding Enclave Page Accesses. At the system level, some lightweight embedded PMAs [34, 27] avoid page table-based threats altogether by implementing hardware-enforced isolation in a single-address-space. Alternatively, some higher-end PMA research prototypes [10, 11, 30, 42] place enclave page tables out of reach of an attacker. Unfortunately, we believe such an approach is unacceptable for Intel SGX, especially when protecting sensitive application data from potentially malicious cloud providers [3, 36]. In such use cases, the cloud provider must be able to quickly regulate different cloud users competing for scarce platform resources including EPC memory. Fortified PMA designs such as Sanctum [10] on the other hand move page tables within the enclave, and require the OS to engage in a lengthy protocol whenever reclaiming a physical page. Furthermore, when applying Sanctum's enclave-private page table design to modern x86 processors [23], an adversary could still leverage the Extended Page Tables (EPTs) set up by the hypervisor. That is, any access to guest-physical pages, including the enclave and its private page tables, results in an EPT walk that sets accessed and dirty bits accordingly. Masking A/D attributes in enclave mode is neither sufficient nor desirable, as it cannot prevent our cache-based attacks, and disrupts benign OS memory management decisions.

At the application level, we believe the academic community should investigate different defense strategies based on the type of enclave. For small enclaves that must be offered the highest security guarantees, automated compiler-based solutions [8] are to be considered. Good practices applied to cryptographic software (e.g., not branching on a secret) may be extended to more general approaches, such as the deterministic multiplexing defense proposed by Shinde et al. [40]. For uses cases where unmodified application binaries are loaded in an enclave, however, such approaches would likely lead to unacceptable performance overhead. In such situations, the use of more probabilistic security measures may be acceptable. Note that previous page fault-driven research [48] successfully defeated conventional Address Space Layout Randomization (ASLR) schemes that randomize an application's base address. SGX-Shield [38], on the other hand, implements fine-grained ASLR by compiling enclaved

application code into small 32- or 64-byte randomization units that can subsequently be re-shuffled at load time.

7 Related Work

A recent line of work has developed PMA security architectures that support secure isolated execution of protected modules with a minimal trusted computing base, either via a small hypervisor [31, 30, 42, 19], or with trusted hardware [29, 32, 11, 10, 34, 27]. Intel SGX represents the first widespread PMA solution, included in off-the-shelf consumer hardware, and has recently been put forward to protect sensitive application data from untrusted cloud providers [3, 36]. As such, SGX has received considerable attention from the research community, and one line of work, including Graphene-SGX [45], Haven [3], Panoply [41], and SCONE [2] has developed small libOSs that facilitate running unmodified legacy applications in SGX enclaves. However, Xu et al. [48] recently pointed out that enclaved execution environments are vulnerable to a new class of powerful controlled-channel attacks conducted by an untrusted host operating system. We have discussed previous research results on page table-based attacks and defenses extensively in Section 2.3. Iago attacks [6] furthermore exploit legacy applications via the system call interface, and AsyncShock [47] demonstrates that an adversarial OS can more easily exploit thread synchronization bugs within SGX enclaves. Finally, between submission and publication of this paper, the SGX research community has witnessed a steady stream of microarchitectural side-channel attacks; either by abusing the branch prediction unit [28], or in the form of fine-grained PRIME+PROBE [13, 37, 5, 33] cache attacks.

In a more general, non-PMA context, there exists a vast amount of research on microarchitectural cache timing vulnerabilities [35, 50, 17]. Especially relevant to our work is the FLUSH+FLUSH [16] channel which was only proposed very recently, and attack research [49] that applies FLUSH+RELOAD to partially recover OpenSSL ECDSA nonces. Furthermore, timing differences from TLB misses have been exploited to break kernel space ASLR [20]. More recently, it has been shown that kernel ASLR can also be bypassed by exploiting timing differences in the `prefetch` instruction [15], or by leveraging TSX [24]. Finally, recent concurrent work [14] on JavaScript environments has independently demonstrated a page table-based cache side-channel attack that completely compromises application-level ASLR.

8 Conclusion

Our work shows that page table walks in unprotected memory leak enclave page accesses to untrusted system software. We demonstrated that our stealthy attack vectors can circumvent current state-of-the-art defenses that hide page faults from the OS. As such, page table-based threats continue to be worrisome for enclaved execution.

Acknowledgments

We thank Ming-Wei Shih for kindly providing us with early access to the camera-ready version of his T-SGX paper. Jo Van Bulck and Raoul Strackx are supported by a grant of the Research Foundation - Flanders (FWO).

References

[1] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy* (2013), vol. 13.

[2] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., ET AL. SCONE: Secure Linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation* (2016), USENIX Association, pp. 689–703.

[3] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), USENIX Association, pp. 267–283.

[4] BERNSTEIN, D. J., DUIF, N., LANGE, T., SCHWABE, P., AND YANG, B.-Y. High-speed high-security signatures. *Journal of Cryptographic Engineering 2*, 2 (2012), 77–89.

[5] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software grand exposure: SGX cache attacks are practical. *arXiv preprint arXiv:1702.07521* (2017).

[6] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013), ACM, pp. 253–264.

[7] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the 12th ACM on Asia Conference on Computer and Communications Security (ASIA CCS)* (2017), ACM, pp. 7–18.

[8] COPPENS, B., VERBAUWHEDE, I., DE BOSSCHERE, K., AND DE SUTTER, B. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 IEEE Symposium on Security and Privacy* (2009), IEEE, pp. 45–60.

[9] COSTAN, V., AND DEVADAS, S. Intel SGX explained. Tech. rep., Computer Science and Artificial Intelligence Laboratory MIT, 2016. `https://eprint.iacr.org/2016/086.pdf`.

[10] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium* (2016), USENIX Association, pp. 857–874.

[11] EVTYUSHKIN, D., ELWELL, J., OZSOY, M., PONOMAREV, D., GHAZALEH, N. A., AND RILEY, R. Iso-x: A flexible architecture for hardware-managed isolated execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), IEEE, pp. 190–202.

[12] FERRAIUOLO, A., WANG, Y., XU, R., ZHANG, D., MYERS, A., AND SUH, E. Full-processor timing channel protection with applications to secure hardware compartments. Computing and information science technical report, Cornell University, November 2015.

[13] GÖTZFRIED, J., ECKERT, M., SCHINZEL, S., AND MÜLLER, T. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)* (2017).

[14] GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., AND GIUFFRIDA, C. ASLR on the line: Practical cache attacks on the MMU. In *24th Annual Network and Distributed System Security Symposium (NDSS)* (2017).

[15] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (2016).

[16] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+flush: A fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2016).

[17] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24nd USENIX Security Symposium* (2015), USENIX Association, pp. 897–912.

[18] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., AND DEL CUVILLO, J. Using innovative instructions to create trustworthy software solutions. In *HASP@ ISCA* (2013), p. 11.

[19] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013), ACM, pp. 265–278.

[20] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 191–205.

[21] INTEL CORPORATION. *Intel Software Guard Extensions Programming Reference*, October 2014. Reference no. 329298-002US.

[22] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, June 2016. Reference no. 248966-033.

[23] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual*, June 2016. Reference no. 325462-059US.

[24] JANG, Y., LEE, S., AND KIM, T. Breaking kernel address space layout randomization with Intel TSX. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (2016), ACM, pp. 380–392.

[25] KOCH, W., AND SCHULTE, M. *The Libgcrypt Reference Manual*, December 2016. Version 1.7.4.

[26] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference* (1996), Springer, pp. 104–113.

[27] KOEBERL, P., SCHULZ, S., SADEGHI, A.-R., AND VARADHARAJAN, V. TrustLite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, pp. 10:1–10:14.

[28] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium* (2017), USENIX Association.

[29] MAENE, P., GÖTZFRIED, J., DE CLERCQ, R., MÜLLER, T., FREILING, F., AND VERBAUWHEDE, I. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 99 (2017).

[30] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V. D., AND PERRIG, A. TrustVisor: Efficient TCB reduction and attestation. In *2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 143–158.

[31] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 2008 EuroSys Conference* (2008), ACM, pp. 315–328.

[32] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), ACM, pp. 10:1–10:1.

[33] MOGHIMI, A., IRAZOQUI, G., AND EISENBARTH, T. Cachezoom: How SGX amplifies the power of cache attacks. *arXiv preprint arXiv:1703.06986* (2017).

[34] NOORMAN, J., VAN BULCK, J., MÜHLBERG, J. T., PIESSENS, F., MAENE, P., PRENEEL, B., VERBAUWHEDE, I., GÖTZFRIED, J., MÜLLER, T., AND FREILING, F. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)* (2017).

[35] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference* (2006), Springer, pp. 1–20.

[36] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. Vc3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 38–54.

[37] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD., S. Malware guard extension: Using SGX to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2017).

[38] SEO, J., LEE, B., KIM, S., AND SHIH, M.-W. SGX-Shield: Enabling address space layout randomization for sgx programs. In *24th Annual Network and Distributed System Security Symposium (NDSS)* (2017).

[39] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *24th Annual Network and Distributed System Security Symposium (NDSS)* (2017).

[40] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS)* (2016), ACM, pp. 317–328.

[41] SHINDE, S., TIEN, D. L., TOPLE, S., AND SAXENA, P. Panoply: Low-TCB linux applications with SGX enclaves. In *24th Annual Network and Distributed System Security Symposium (NDSS)* (2017).

[42] STRACKX, R., AND PIESSENS, F. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)* (2012), ACM, pp. 2–13.

[43] TRAMER, F., ZHANG, F., LIN, H., HUBAUX, J.-P., JUELS, A., AND SHI, E. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2nd IEEE European Symposium on Security and Privacy (Euro S&P)* (2017), IEEE.

[44] TSAI, C.-C., ARORA, K. S., BANDI, N., JAIN, B., JANNEN, W., JOHN, J., KALODNER, H. A., KULKARNI, V., OLIVEIRA, D., AND PORTER, D. E. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 9.

[45] TSAI, C.-C., PORTER, D. E., AND VIJ, M. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC)* (2017), USENIX Association.

[46] WANG, W., CHEN, G., PAN, X., ZHANG, Y., WANG, X., BINDSCHAEDLER, V., TANG, H., AND GUNTER, C. A. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. *arXiv preprint arXiv:1705.07289* (2017).

[47] WEICHBRODT, N., KURMUS, A., PIETZUCH, P., AND KAPITZA, R. Asyncshock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security (ESORICS)* (2016), Springer.

[48] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 640–656.

[49] YAROM, Y., AND BENGER, N. Recovering OpenSSL ECDSA nonces using the flush+ reload cache side-channel attack. *IACR Cryptology ePrint Archive 2014* (2014), 140.

[50] YAROM, Y., AND FALKNER, K. Flush+reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium* (2014), USENIX Association, pp. 719–732.

A Libgcrypt Page Sets

For completeness, we provide the full page sets for the different versions of our Libgcrypt attacks below. The PTE sets are based on plain Libgcrypt, Libgpg-error, and Graphene-libc binaries, as generated by `gcc` v5.2.1 from the default `./configure && make` invocation. The provided addresses are relative to the load addresses used by Graphene, as explained in Section 4.

```
#if CONFIG_SPY_GCRY && (CONFIG_SPY_GCRY_VERSION == 163)
    #define SET_ADRS        (GCRYLIB_ADRS + 0xa7780) // _gcry_mpi_set
    #define TST_ADRS        (GCRYLIB_ADRS + 0xa0a00) // _gcry_mpi_test_bit
    #define MULP_ADRS       (GCRYLIB_ADRS + 0xa97c0) // _gcry_mpi_ec_mul_point
    #define TDIV_ADRS       (GCRYLIB_ADRS + 0xa1310) // _gcry_mpi_tdiv_qr
    #define ERR_ADRS        (GPG_ERR_ADRS + 0x0b6d0) // gpg_err_set_errno
    #define FREE_ADRS       (GCRYLIB_ADRS + 0x0ce90) // _gcry_free
    #define PFREE_ADRS      (GCRYLIB_ADRS + 0x110a0) // _gcry_private_free
    #define XMALLOC_ADRS    (GCRYLIB_ADRS + 0x0d160) // _gcry_xmalloc
    #define MUL_ADRS        (GCRYLIB_ADRS + 0xa6920) // _gcry_mpih_mul
    #define PMALLOC_ADRS    (GCRYLIB_ADRS + 0x10f80) // _gcry_private_malloc

    #define MONITOR_ADRS    SET_ADRS

    void construct_pte_set(spy_pte_set_t *set)
    {
        pr_info("gsgx-spy: constructing A/D PTE set for gcry v1.6.3\n");
        add_to_pte_set(set, TST_ADRS);
        add_to_pte_set(set, MULP_ADRS);
        add_to_pte_set(set, TDIV_ADRS);
        add_to_pte_set(set, ERR_ADRS);
        add_to_pte_set(set, FREE_ADRS);
        add_to_pte_set(set, PFREE_ADRS);
        add_to_pte_set(set, XMALLOC_ADRS);
        add_to_pte_set(set, MUL_ADRS);
        add_to_pte_set(set, PMALLOC_ADRS);
    }

#elif CONFIG_SPY_GCRY && (CONFIG_SPY_GCRY_VERSION == 175)
   #if CONFIG_FLUSH_FLUSH
        #define ERRNOLOC_ADRS   (LIBC_ADRS + 0x20590)    // __errno_location
        #define MULP_ADRS       (GCRYLIB_ADRS + 0xca220) // _gcry_mpi_ec_mul_point
        #define TST_ADRS        (GCRYLIB_ADRS + 0xc10d0) // _gcry_mpi_test_bit
        #define _GPGRT_ADRS     (GPG_ERR_ADRS + 0x2bb0)  // _gpgrt_lock_lock
        #define GPGRT_ADRS      (GPG_ERR_ADRS + 0xb750)  // gpgrt_lock_lock
        #define INT_FREE_ADRS   (LIBC_ADRS + 0x7b110)    // _int_free
        #define PLT_ADRS        (GCRYLIB_ADRS + 0xab30)  // __errno_location@plt
        #define DO_MALLOC_ADRS  (GCRYLIB_ADRS + 0xe380)  // do_malloc

        #define MONITOR_ADRS    ERRNOLOC_ADRS

        void construct_pte_set(spy_pte_set_t *set)
        {
            pr_info("gsgx-spy: constructing F+R PTE set for gcry v1.7.5\n");
            add_to_pte_set(set, MULP_ADRS);
            add_to_pte_set(set, TST_ADRS);
            add_to_pte_set(set, _GPGRT_ADRS);
            add_to_pte_set(set, GPGRT_ADRS);
            add_to_pte_set(set, INT_FREE_ADRS);
            add_to_pte_set(set, PLT_ADRS);
            add_to_pte_set(set, DO_MALLOC_ADRS);
        }

   #else /* !CONFIG_FLUSH_FLUSH */
        #define TST_ADRS        (GCRYLIB_ADRS + 0xc10d0) // _gcry_mpi_test_bit
        #define ADDP_ADRS       (GCRYLIB_ADRS + 0xc9bc0) // _gcry_mpi_ec_add_p
        #define MULP_ADRS       (GCRYLIB_ADRS + 0xca220) // _gcry_mpi_ec_mul_p
        #define FREE_ADRS       (GCRYLIB_ADRS + 0x0f390) // _gcry_free
        #define ADD_ADRS        (GCRYLIB_ADRS + 0xc0a10) // _gcry_mpi_add

        #define MONITOR_ADRS    TST_ADRS

         void construct_pte_set(spy_pte_set_t *set)
         {
             pr_info("gsgx-spy: constructing A/D PTE set for gcry v1.7.5\n");
             add_to_pte_set(set, ADDP_ADRS);
             add_to_pte_set(set, MULP_ADRS);

             add_to_pte_set(set, FREE_ADRS);
             add_to_pte_set(set, ADD_ADRS);
         }
   #endif
#endif
```