On the Challenges of Detecting Side-Channel Attacks in SGX

Jianyu Jiang
The University of Hong Kong
Hong Kong, China
jyjiang@cs.hku.hk

Claudio Soriente
NEC Laboratories Europe
Heidelberg, Germany
claudio.soriente@neclab.eu

Ghassan Karame
Ruhr-University Bochum
Bochum, Germany
ghassan@karame.org

Existing tools to detect side-channel attacks on Intel SGX are grounded on the observation that attacks affect the performance of the victim application. As such, all detection tools monitor the potential victim and raise an alarm if the witnessed performance (in terms of runtime, enclave interruptions, cache misses, etc.) is out of the ordinary.

In this paper, we show that monitoring the performance of enclaves to detect side-channel attacks may not be effective. Our core intuition is that all monitoring tools are geared towards an adversary that interferes with the victim's execution in order to extract the most number of secret bits (e.g., the entire secret) in one or few runs. They cannot, however, detect an adversary that leaks smaller portions of the secret—as small as a single bit—at each execution of the victim. In particular, by minimizing the information leaked at each run, the impact of any side-channel attack on the application's performance is significantly lowered—ensuring that the detection tool does not detect an attack. By repeating the attack multiple times, each time on a different part of the secret, the adversary can recover the whole secret and remain undetected. Based on this intuition, we adapt known attacks leveraging page-tables and L3 cache to bypass existing detection mechanisms. We show experimentally how an attacker can successfully exfiltrate the secret key used in an enclave running various cryptographic routines of `libgcrypt`. Beyond cryptographic libraries, we also show how to compromise the predictions of enclaves running decision-tree routines of OpenCV. Our evaluation results suggest that performance-based detection tools do not deter side-channel attacks on SGX enclaves and that effective detection mechanisms are yet to be designed.

1

Intel Software Guard Extensions (SGX) enables applications to execute in isolation from other software on the same platform, including the OS. SGX-enabled processors run applications in so-called *enclaves* and provide them with encrypted runtime memory, encrypted storage, and mechanisms to issue authenticated statements on the enclave software configuration. As such, a number of practitioners believe that Intel SGX is particularly suited for cloud deployments since it allows to outsource applications to the cloud, with the assurance that outsourced applications run untampered and their data is not available to any (privileged) software on the same host.

Previous work has, however, shown that Intel SGX exhibits a number side-channels that, when coupled with an adversary that controls the OS, allow for effective leakage of enclave secrets [9, 20, 25, 32, 34]. Alongside attacks, the research community has proposed a number of prevention [5, 8, 12, 16, 27] and detection mechanisms [14, 23, 26]—the former having usually much higher overhead compared to the latter. To the best of our knowledge, all detection mechanisms are grounded on the observation that side-channel attacks affect the performance of the victim application (e.g., by increasing the number of enclave interruptions) and, therefore, signal an attack when the witnessed performance is anomalous.

In this paper, we show that such detection tools may not be effective at detecting side-channel attacks on SGX enclaves. Namely, existing detection mechanisms are geared towards an adversary that interferes with the victim's execution in order to extract the most number of secret bits (e.g., the entire secret) in one or few runs. Such an attack strategy has a significant impact on the victim's performance, effectively allowing detection mechanisms to notice a deterioration in performance (e.g., in terms of runtime, enclave interruptions, cache misses, etc.) and signal an attack.

Our core intuition is that an adversary can leak smaller portions of the secret—as small as a single bit—at each execution of the victim, so as to minimize the impact on its performance and, therefore, remain undetected. More specifically, we show that an adversary can profile a victim enclave, thereby identifying the precise moment during the victim's execution when a specific part of the secret can be leaked via a side-channel attack. For example, if the victim runs the popular square-and-multiply algorithm, we show that the attacker can infer the moment when the i-th loop is being executed—i.e., when the i-th secret bit is being processed—and execute a side-channel attack at that time to leak the secret bit, without affecting the performance of the victim. By running the victim multiple times and leaking a different part of the secret at a time, our technique can recover the whole secret while remaining undetected.

Based on this intuition, we adapt known attacks leveraging page-tables, L3 cache, and a combination of the two, and evaluate their performance on routines of `libgcrypt` (namely, `mpi_powm` and `mpi_ec_dup_point`) used by popular cryptographic primitives such as ElGamal, RSA, and EdDSA. We also apply our attack strategy on non-cryptographic software and evaluate how to leak predictions of enclaves running decision-tree routines of OpenCV [3]. Our results show that our strategy recovers up to 100% of a secret key used in `libgcrypt` routines, depending on the type of side-channel exploited, and with marginal impact on the victim's performance (as low as one extra Asynchronous EXit (AEX) or roughly 40 cache misses per run). In case of a victim using the decision-tree routines of OpenCV to predict handwritten digits of the MNIST data-set [2], our attack strategy can correctly leak around 55% of the predictions (whereas a "standard" side-channel attack, that is easily detected by available tools, reaches 64% of leaked predictions).

We additionally show that an adversary using our attack strategy cannot be detected by existing detection tools such as T-SGX [26], unless one tolerates a large number of false positives. We also provide evidence that *any* detection tool that monitors the performance of the victim is equally likely to fail. We do so by assuming a comprehensive tool (dubbed Monitor++) that monitors all of the performance metrics proposed in literature and show that even such a tool cannot distinguish between a benign and a "malicious" execution.

Our results highlight that defenses that monitor performance metrics are not enough to detect side-channel attacks on Intel SGX enclaves. We therefore hope that our findings help avoiding additional (and probably unnecessary) cycles of defenses that monitor performance metrics and attacks that succeed at bypassing them.

The rest of this paper is organized as follows. In Section 2, we overview necessary background information and related work on SGX. We describe the main intuition behind our attacks in Section 3 and we evaluate them against `libgcrypt` and OpenCV in Sections 4-6. Finally, Section 7 discusses possible defenses against our attacks and provides some concluding remarks.

2 [redacted]

Previous research has shown that Intel SGX is vulnerable to side-channel attacks and that the Intel SGX threat model—by considering a malicious OS—allow for very effective attacks [9, 20, 21, 31].

Proposed defenses work either as prevention or detection tools. Prevention techniques incur in high overhead [5, 8, 16, 27], and sometimes can only prevent specific types of side-channels [12].

Detection techniques have usually lower overhead and, to the best of our knowledge, they all use the same "anomaly-based" approach: they monitor the execution of the victim application and signal an attack in case of deviations from a "normal" execution. Varys [23] prevents L1/L2 cache-based attacks with core-reservation; at the same time, Varys detects attacks based on page-faults or interrupts by monitoring the number of AEXs so that an alarm is raised if their frequency is too high. Varys is currently part of a commercial product and its source-code is not available. Déjà Vu [14] detects attacks based on page-faults or interrupts by monitoring the execution time of the enclave. Déjà Vu instruments the basic blocks of the enclave code to measure their execution time and an attack is "detected" if the total time deviates from the one of an execution in a benign environment. An incomplete version of Déjà Vu is available on github [13]; we made contact with the authors to obtain the missing code, but they are no longer maintaining the project. T-SGX [26] makes use of Transactional Synchronization eXtensions (TSX) to suppress page-faults notifications to the OS. When an interrupt or fault is thrown within a TSX transaction, T-SGX aborts and executes a user-defined handler. The handler of T-SGX keeps tracks of the number of aborts per transaction and raises an alarm if that number reaches a given threshold. The source code of T-SGX is available on github [29].

[redacted] Previous work proposes side-channel attacks on enclaves that do not cause page-faults—thereby achieving stealthiness despite detection-tools that monitor page-faults. Jo Van et al., [11] monitor the `ACCESS` bit of the page-table to get the page access sequence of the victim without page-faults. As the `ACCESS` bit of a page-table is set only the first time the page is accessed (i.e., subsequent accesses do not modify the bit), the authors of [11] force a TLB shootdown—by interrupting the enclave via inter-process-interrupts—to reset the `ACCESS` bit. The authors acknowledge that the number of interruptions during their attack is substantially higher than what is to be expected under benign circumstances, and suggest that a detection tool may notice the attack by monitoring enclave interruptions rather than page-faults. Differently, the attack strategy we develop in this paper causes only a few interruptions of the victim and remains undetected.

Wang et al., [32] show that enclave interruptions can be minimized if TLB shootdown is achieved by using a sibling hyperthread that probes memory addresses whose TLB entries are conflicted with the ones of the victim enclave. While the attack developed by [32] cannot be detected by monitoring enclave interruptions, it requires the adversary and the victim to run on the same core. As such, the attack is not viable in case of detection tools that enforce core-reservation like Varys [23] or Déjà Vu [14]. Differently, our attack strategy does not require the adversary to run on the same core of the victim.

Another stealthy attack is Prime+Abort [15]. Here, the idea is to use TSX as a "watchdog" so that whenever the victim touches a specific cache-line, the adversary receives an immediate hardware callback in the form of a transactional abort. In principle, the attack could be used to bypass detection mechanism that monitor asynchronous exits [14, 23] or that use TSX to suppress page-faults notifications to the OS [26]. However, a Prime+Abort attack could be easily spotted by monitoring cache misses. In Section 5, we show that a trivial modification to T-SGX [26] allows a victim to detect Prime+Probe attacks.

3 [redacted]

3 [redacted]

We assume that the adversary has the victim code available (e.g., the code belongs to a library or an open-source implementation), controls the OS where the enclave is running, and can execute the victim enclave arbitrarily many times. Such assumptions are similar to the ones found in related work [20, 32, 34] and capture a realistic cloud deployment where an application is uploaded by its owner to the cloud provider, and part of the application code (e.g., a decryption routine) runs in an enclave. After attestation and secret provisioning by the application owner, the cloud provider can (re-)start the application or trigger the routine running in the enclave arbitrarily many times.

3 [redacted]

Detection tools for (known) side-channel attacks build on the intuition that attacks are likely to alter the performance of the victim application. As a consequence, almost all detection tools monitor the performance of the potential victim, and signal an attack if the witnessed performance is anomalous.

We show that this intuition is not accurate. More precisely, we show that an adversary can bypass these tools while minimizing the effect on the victim's performance by "spreading" the attack across multiple runs. This can be done when the adversary extracts specific portions of the secret, as small as a single bit, at each run of the victim enclave. By minimizing the information leaked at each run, the impact of the attack on the victim's performance is also lessened—so that the detection tool notices no performance anomaly. This strategy is repeated for a number of times—each time leaking a different portion of the secret—to eventually recover the full secret.

In particular, we denote the enclave secret by $\mathtt{s} = \mathtt{s}_1,...,\mathtt{s}_n$, where each $\mathtt{s}_i$ could be a single bit or multiple ones. Moreover, assume the victim code is split into n segments $S_1,...,S_n$, such that segment S_i processes $\mathtt{s}_i$. Here, the application is executed n times. During the i-th run, the attacker launches a side-channel attack while the victim is executing segment S_i, in order to leak $\mathtt{s}_i$. As the attack

only runs for a small time-window, the victim's performance is only marginally affected.

Developing the aforementioned strategy entails a number of challenges and requires the adversary to mount a side-channel attack only during the time-window when the victim is executing code segment S_i. One option would be to precisely control the victim's execution by using single-stepping frameworks like SGX-Step [31]. However, side-stepping the victim generates a large number of page-faults—allowing a tool that monitors the number of AEXs to detect such an attack. To remedy this, we take a different approach and design an offline automated profiling phase to learn the time-interval when the victim is executing a specific code segment S_i. In what follows, we detail the offline profiling phase and the design choices we made to minimize errors.

3

Let T_i be the time when the victim starts code segment S_i. Note that a segment is a logical execution unit and different segments may execute the same code, but on different portions of the secret. For example, in the square-and-multiply routine, each segment corresponds to one execution of the main loop and processes one secret bit.

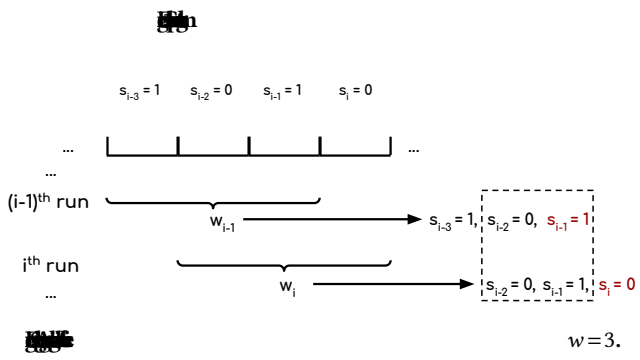
In an ideal scenario, the execution time of each code segment is constant, i.e., $T_{i+1} - T_i = c$. Thus, segment S_i starts at time $T_i = (i-1)\cdot c$, for some constant c. More generally, the execution time of a code segment may depend on the code itself, as well as the portion of the secret it processes. Thus, we model the execution time of segment S_i as a function $t_i(\mathtt{s}_i)$, and set the start time of segment S_i as $T_i = \sum_{j<i} t_j(\mathtt{s}_j)$.

As an example, Figure 1 shows a simple code segment with a conditional branch on the i-th bit of variable `secret` and three different function calls (`m`, `g`, and `k`). If functions `m`, `g` and `k` have no conditional branches nor loops, we can use constants $c_\mathtt{m}$, $c_\mathtt{g}$, and $c_\mathtt{k}$, to model their execution time. Thus, $t_i(\mathtt{s}_i) = c_\mathtt{m} + \mathtt{s}_i \cdot c_\mathtt{g} + (1-\mathtt{s}_i)\cdot c_\mathtt{k}$. In case any of the functions `m`, `g`, `k` has a loop or a conditional branch, we would recursively profile its execution time in a similar fashion.

Once we have the function $t_i(\mathtt{s}_i)$ that models the execution of S_i, we assess its values by running S_i multiple times, and by using a different assignment of $\mathtt{s}_i$ each time. For example, to measure the execution time of the code in Figure 1, we run the segment twice: once with $\mathtt{s}_i = 0$ and once with $\mathtt{s}_i = 1$. (We actually use multiple runs with the same configuration of variables, in order to make our measurements more robust.) Note that the enumeration of all possible configurations of the part of the secret processed by a code segment is feasible because each segment is likely to process only one or a few secret bits.

[illegible] Since SGX cannot read the time-stamp counter via `rdstc`, we cannot directly inject time measurement instructions into code segments. Furthermore, when measuring the execution time of each code segment we cannot interrupt the enclave, as context switches between enclave and non-enclave code incurs extra overhead compared to context switches between regular processes [7, 33]. We, therefore, create a logical clock by means of a timer thread. We inject instructions at the start and end of each segment, to set a binary variable at a memory address `Addr` outside of the enclave memory. A separate timer thread continually gets the system timestamp using `rdstc` and checks the value of the variable at `Addr`. If the variable is set to 1, the timer thread remembers the current timestamp and reset `Addr` to `0`. By measuring the time interval between two reads of `Addr` that returned 1, we can infer the time required to run one code segment.

```
1  void compute_on_s(char[] p, unsigned int secret) {
2   int tmp = m(p);
3   for (int i = 0;i < NBITS;i++) {
4    if ((secret & (1 << i)) != 0)
5     g(tmp);
6    else
7     k(tmp);
8   }
9  }
```



[illegible] Running time of arbitrary code on general-purpose machines is far from deterministic due to other software running on the same host. Similar to [32], we reduce the noise due to other software on the same host by reserving a core for the victim enclave. Specifically, we use the `isolcpus` as boot-up option in Ubuntu. As a result, no tasks are assigned to the reserved core, nor it is interrupted for handling I/O. Furthermore, processes can be explicitly assigned to such cores (e.g., using `sched_setaffinity`) and they can be interrupted by inter-processor interrupts. We also note that some detection tools [14, 23] ensure core reservation to avoid side-channel attacks based on L1/L2 caches.

To reduce the noise due to state of the cache when the victim starts, we flush all caches before each execution. Although techniques such as speculative execution may still create differences in the state of the caches across different executions of the enclave, we have empirically verified that each run experiences almost the same amount of cache misses. We also disable dynamic frequency scaling and fix the CPU frequency to stabilize execution time.

By combining core-reservation with cache-flushing and a fixed CPU frequency, we manage to stabilize the execution time of the victim (i.e., within 0.1%).

[illegible] The accuracy of our technique relies on the correct estimation of T_i—when we start the side-channel attack to learn $\mathtt{s}_i$—and the correct guess of $\mathtt{s}_i$.

Clearly, an error when estimating T_i leads to a mis-alignment between the attack and the victim that, in turn, leads to unpredictable errors in inferring the secret $\mathtt{s}_i$. An error when inferring $\mathtt{s}_i$ may lead to an error in the estimation of T_j for $j > i$ since the start time of segment S_j may depend on the value of $\mathtt{s}_1,\ldots,\mathtt{s}_{j-1}$.

One possible option to avoid miss-alignments between the victim and the attack is to rely on deterministic signals thrown by the enclave such as page-faults, page ACCESS bit, TSX aborts and so on. For example, the attacker may invalidate a page that is required by the victim at the start of S_i so that a page-fault is thrown when the victim starts executing that segment. Alternatively, alignment errors may be corrected by attacking multiple consecutive segments at a time by using a sliding window. This basic idea is shown in Figure 2. Let w_i be the window attacking segments $S_{i-w+1},...,S_i$ so to obtain bits $s_{i-w+1},...,s_i$ and, without loss of generality, assume the step of the window to be 1. Then, we compare the guess for bits $s_{i-w+1},...,s_{i-1}$ obtained when attacking window w_i, with the guess for the same bits obtained when attacking window w_{i-1} (i.e., when attacking segments $S_{i-w},...,S_{i-1}$). If the two bit sequences match, then we assume that window w_i is well aligned and treat the guess for the last bit of the window (i.e., s_i) as valid; otherwise we assume w_i is not aligned with the victim and discard the guess for s_i.

Note that attacking larger windows may have an impact on the victim's performance that could allow a detection mechanism to spot the attack. Also, our attack with $w = n$ becomes similar to a "standard" side-channel attack that tries to leak all secret bits at once.

In order to improve the accuracy of our technique, we can also increase the number of times we attack a given segment. That is, we run the victim k times and run the attack on the same segment S_i (or segment window w_i). We therefore obtain several samples for s_i and use heuristics to improve the accuracy of our guess.

[redacted] To automate the profiling process, we expose two macros, `SEGMENT_START(secret)` and `SEGMENT_END`, for annotating the start and end of one segment, along with the portion of the secret consumed by that segment. These macros are then compiled with the victim code to generate a corresponding time-measuring code that records the execution time. During the profiling process, the portion of the secret used by a segment—usually one or a few bits—is enumerated and fed to the time-measuring code. For each configuration of the secret value, the time-measuring code generates a report with the execution time. The profiling process repeats to collect sufficient reports for stably modeling execution time of the victim.

4 [redacted] LIBGCRYPT

We now show how to instantiate the strategy described earlier on cryptographic routines of `libgcrypt`, namely `mpi_powm` (used in ElGamal, RSA, and DSA) and `mpi_ec_mul_point` (used in EdDSA). We leverage a side-channel based on time [18, 32], one based on memory access pattern [11, 27], and a combination of the two. In what follows, we use `libgcrypt` version `1.7.0`; the side-channels we exploit are present in `mpi_powm` up to version `1.8.6`, and in `mpi_ec_mul_point` up to version `1.7.5`. We stress however that the above side-channels are mere examples to showcase the effectiveness of our strategy; our techniques are independent of the underlying side-channel and could use any other workable side-channel.

4.1 [redacted] `mpi_powm`

Figure 3 shows the code of `mpi_powm`. The routine has two side-channels, one based on time and another based on memory access pattern.

```
1  void _gcry_mpi_powm (gcry_mpi_t res, gcry_mpi_t base,
2                       gcry_mpi_t expo, gcry_mpi_t mod) {
3    /* ... */
4    gcry_mpi_t e = expo;
5    int esec  = mpi_is_sec(expo);
6    for (;e != 0;e = (e << 1)) {
7      _gcry_mpih_sqr_n_basecase( xp, rp, rsize );
8      if(esec || (mpi_limb_signed_t)e < 0 ) {
9        /*mpihelp_mul( xp, rp, rsize, bp, bsize );*/
10       if( bsize < KARATSUBA_THRESHOLD ) {
11         _gcry_mpih_mul ( xp, rp, rsize, bp, bsize );
12       } else {
13         _gcry_mpih_mul_karatsuba_case (xp, rp, rsize,
14                                        bp, bsize,
15                                        &karactx);
16       }
17       xsize = rsize + bsize;
18       if ( xsize > msize ) {
19         _gcry_mpih_divrem(xp + msize, 0, xp,
20                           xsize, mp, msize);
21         xsize = msize;
22       }
23       if ( (mpi_limb_signed_t)e < 0 ) {
24         tp = rp; rp = xp; xp = tp;
25         rsize = xsize;
26       }
27     }
28   }
29 }
```

[redacted] `mpi_powm` **[redacted]**

The loop (line 6 ~ 28) consumes one bit of the secret exponent per iteration and executes an extra computation (line 9 ~ 27) if that bit is 1 (line 8). Thus, an adversary can infer the secret bit of the exponent being processed, by inferring the time to complete one loop iteration. Note that if `esec` is 1, then the exponent is stored in secure memory, and the conditional branch is always executed to eliminate side-channels. However, if `xvalue` is provided as input by the user (e.g., when the key-pair is generated from a passphrase), then `libgcrypt` does not store the exponent in secure memory so that side-channels are not eliminated.

Alternatively, the secret bit of the exponent can be leaked by monitoring access to memory pages that store the code required by the `if`-branch of the routine. Let `A`, `B`, `C` be the addresses of `mpi_powm`, `mpi_mpih_sqrt_n_basecase` and `mpihelp_mul`, respectively. One iteration of the loop where the exponent bit is 1, shows a memory access sequence like `ABCAC|AB`, whereas if the exponent bit is `0`, the observed memory access sequence is like `ABC|AB`. In these examples, memory accesses after `|` belong to the next iteration of the loop. Also, note that `mpi_mpih_sqrt_n_basecase` calls `mpihelp_mul`, so there will always be an access to address `C` after `B`. One could infer the memory access sequence either by observing page-faults or cache accesses.

4.2 [redacted] `mpi_powm`

In order to profile `mpi_powm`, we define each iteration of the main loop as one segment. Let s_i be the i-th exponent bit consumed in segment S_i. One iteration of the loop in `mpi_powm` computes on `xp`,

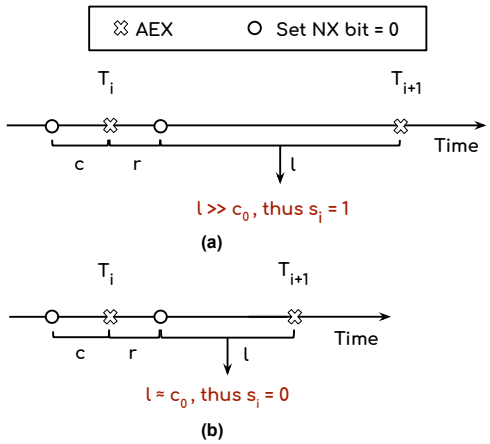


Figure 4: Our-PF attack strategy. (a) $s_i = 1$. (b) $s_i = 0$.

`rp` and s_i. We found that all branches and loops in `mpih_sqr_n_basecase` have negligible impact on execution time, so we consider its runtime constant and we denote it by c_{base}. Thus, runtime of S_i with $s_i = 0$ is simply c_{base}. If $s_i = 1$, the code executed (lines 9 ~ 27) has two branches. The first one is a conditional branch that, depending on the value of `bsize`, may run either `mpihelp_mul` or `mpihelp_mul_karatsuba_case`. We found that both paths take the same time so we model this time as a constant $c_{mpihelp}$. The second branch depends on `xsize` and `msize`. However, we found that the time taken to run `mpihelp_divrem` is negligible, so we just ignore it. In a nutshell, the time to run segment S_i is $t_i(s) = c_{base} + s_i \cdot c_{mpihelp}$ and $T_i = (i-1) \cdot c_{base} + \sum_{j<i} s_j \cdot c_{mpihelp}$.

4 [illegible]

We start by describing an instantiation of our attack strategy that only uses page-faults and that leverages the timing side-channel of the victim; we denote this attack variant as Our-PF. More specifically, for $i = 1,\ldots,n$, we start the victim enclave running `mpi_powm`, and use a single page-fault to stop it at the beginning of segment S_i. Next, we resume the victim and measure—again, using one page-fault—the time it takes to complete that segment, in order to learn the secret bit s_i.

Figure 4 shows how the Our-PF attack strategy works. Assume the time it takes to run one iteration of the loop with exponent bit `0` and 1 is c_0 and c_1 (with $c_0 < c_1$), respectively. We start the victim and set the `NX` bit of the page containing `mpih_sqr_n_basecase`, right before time T_i (i.e., at $T_i - c$, for some small constant c). As a result, the victim stops and throws a page-fault at the beginning of segment S_i—i.e., at the beginning of the i-th iteration of the loop. At this time, we resume the victim enclave, and sets again the bit `NX` of the page of `mpih_sqr_n_basecase`. Therefore, the next page-fault will be thrown when the victim moves to the next segment. Hence, the time between the two page-faults is compared against c_0 and c_1, to decide the value of bit s_i. Once we learn s_i, we compute T_{i+1} accordingly and move on to attack the next segment. This process is repeated for $i = 1,\ldots,n$ in order to recover the whole secret. In practice, we also make sure that the `NX` bit is not set while the victim is running `mpih_sqr_n_basecase`. We do so by ensuring that the bit is set r ticks after the page-fault, where r is the number of ticks required to run `mpih_sqr_n_basecase`.

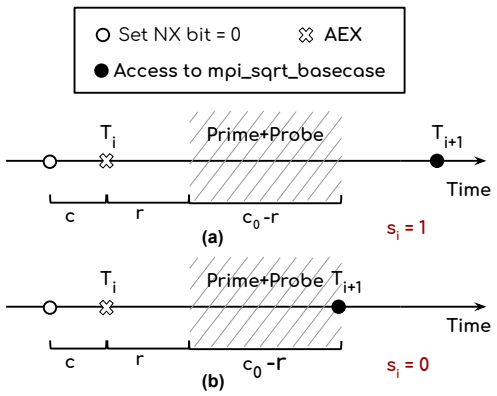


Figure 5: [illegible] $s_i = 1$ [illegible] $s_i = 0$.

4 [illegible]

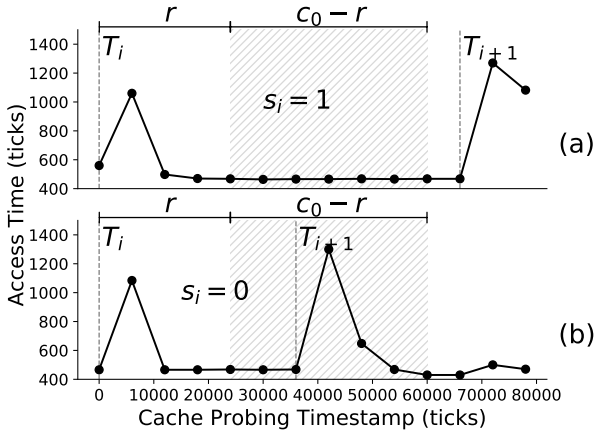
We now describe another attack variant, called Our-PFCa, that leverages both page-faults and cache misses. As Our-PFCa only leverages a side-channel based on memory access pattern, it could be used on routines that have no side-channel based on time.

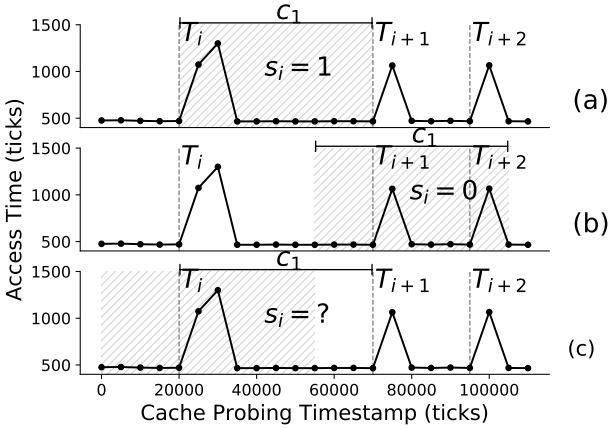
Similar to Our-PF, we use one page-fault to stop the enclave at the beginning of a segment. Then, we use a Prime-and-Probe attack on L3 cache to infer the secret bit processed during the execution of that segment. We use L3 since most detection tools prevent L1/L2 attacks by occupying the entire core.

The workflow of Our-PFCa is shown in Figure 5. Let c_0 and c_1 (with $c_0 < c_1$) be the time it takes to run one loop of `mpi_powm` with secret bit `0` and 1, respectively. We stop the enclave at $T_i - c$ by making the page of `mpih_sqr_n_basecase` unavailable at that time; next, we resume the victim and wait for r clock ticks to make sure that computation on `mpih_sqr_n_basecase` is over. Now, the goal is to measure whether the next call to `mpih_sqr_n_basecase` happens after time $c_0 - r$ or $c_1 - r$. To do so, we start a Prime-and-Probe attack on the address of `mpih_sqr_n_basecase`, for a period of $c_0 - r$. We construct the eviction set of the Prime-and-Probe using techniques from previous research [19]. Figure 6 shows the time to access the target cache set when the secret bit is 1 (a) or `0` (b). Here, it is clear that the victim has accessed the cache line of `mpih_sqr_n_basecase` if the access time of the attacker to the eviction set is larger than 1000 ticks. The first peak in each figure denotes the start of the i-th iteration, while the shaded area denotes the interval of $c_0 - r$ ticks during which we run the Prime-and-Probe attack. Note that if s_i is 1 (Fig. 6a) we do not witness any access to `mpih_sqr_n_basecase` while running the Prime-and-Probe attack. In case s_i is `0` (Fig. 6b) we witness access to `mpih_sqr_n_basecase` as the routine moves to the next iteration of the loop. Once we learn s_i, we compute T_{i+1} accordingly, and move on to attack the next segment.

5 [illegible]

The attack strategies above use page-faults to temporally align the victim and attack threads. We now show how to run cache-only





attacks on the victim enclave. In the sequel, we refer to this strategy as Our-Ca.

Note that, using only cache to leak a specific portion of the victim's secret may be difficult because the adversary thread may not be aligned with the one of the victim; nevertheless, Our-Ca is particularly effective with detection tools that monitor enclave exits (AEXs) [24, 26] as it enables the leakage of the secret without interrupting the victim at all.

Our-Ca works by starting a Prime-and-Probe attack on the address of `mpih_sqr_n_basecase` right before T_i and for c_1 ticks—the number ticks required to complete the loop iteration when the secret bit is 1. If the attack thread experiences a peak in the time to access the target cache set, followed by a sufficient number of lows, we conclude that $\mathtt{s}_i = 1$, whereas if the attack thread experiences two close peaks, we conclude that $\mathtt{s}_i = 0$.

A considerable challenge when using Our-Ca lies in the fact that small errors when estimating T_i leads to unpredictable cache patterns. This is shown in Figure 7. In Figure 7(a), the attack starts at the right time and the witnessed cache pattern does indeed support a correct guess of $\mathtt{s}_i$. Differently, in Figure 7(b) the attack starts late and the adversary (mistakenly) estimates $\mathtt{s}_i$ to be `0`. Finally, in Figure 7(c), the attack starts early, preventing the adversary from estimating the value of the secret bit.

We correct alignment errors between adversary and victim by using a sliding window technique as explained in Section 3. That is, when attacking window w_i (i.e., segments $S_{i-w+1},...,S_i$) we start the Prime-and-Probe attack right before T_{i-w+1} and we run it for wc_1 ticks—i.e. until the end of segment S_i. Next, we consider the estimate of $\mathtt{s}_i$ as valid only if the estimate of $\mathtt{s}_{i-w+1},...,\mathtt{s}_{i-1}$ matches the estimate of the same bits when attacking window w_{i-1}. Finally, we also repeat the attack on the same window a number $k \geq 1$ of times in order to obtain multiple guesses for the same bit and use an heuristic to infer its actual value.

4 Attacking `mpi_ec_mul_point`

We now briefly discuss how to adapt our attack strategy to the `mpi_ec_mul_point` routine of `libgcrypt` used in EdDSA. For each signature, this subroutine is used to compute scalar multiplication with a nonce that, if leaked, allows the recovery of the signing key. Note that our attack extracts one secret bit for each execution of the victim; hence, if the victim picks a fresh nonce at each execution, two bits extracted by our attack would be completely uncorrelated. Nevertheless, EdDSA is deterministic [30] and the nonce is computed as function of the message to be signed and the signing key. Hence, by feeding a fixed message to the signing routine we ensure that the nonce is always the same and can extract one of its bits at each execution.

Figure 8 shows the code of `mpi_ec_mul_point`. Note that the same routine is used to process both the nonce and the signing key (referred to as `scalar` in both cases). The leakage-free code (line `4` ∼ `6`) is used when processing the singing key, whereas the `else`-branch is taken to process the nonce. In the latter case, a secret-dependent branch (line `10`) can be abused to leak one bit of the (secret) nonce. Once the nonce and the corresponding signature are available, the signing key can be computed.

`mpi_ec_mul_point`. Let segment S_i be the i-th iteration of the loop. We found that there are no conditional loops nor branches in `gcry_mpi_ec_dup_point` that have noticeable impact on execution time, so we model its execution time with constant c_{base}. In case the bit of `scalar` being processed is `1`, the routine calls another constant-time function called `mpi_ec_add_point` and we model its execution time with constant c_{add}. Therefore, the running time of the i-th loop iteration is $t_i(\mathtt{s}) = c_{base} + s_i \cdot c_{add}$, and the start time of the i-th segment is $T_i = (i-1) \cdot c_{base} + \sum_{j<i} s_j \cdot c_{add}$.

In practice, we must also accommodate for the first iteration of the loop that takes the `if`-branch; this iteration must fetch `mpi_ec_add_point` and its callees from the main memory and incurs in a time increase that we model with c_{miss}. Thus the start time for the i-th segment becomes $T_i = (i-1) \cdot t_{base} + \sum_{j<i} s_j \cdot c_{add} + (\sum_{j=1}^{j=i-1} s_i \bmod 2) \cdot t_{cache\_miss}$. This extra time for the first loop that processes a 1 bit does not show in `mpi_powm`, as the secret-dependent call to `mpi_sqrt_n_basecase` is also called in other function before `mpi_powm`.

When attacking `mpi_ec_mul_point` we use the page with `mpi_ec_dup_point` to stop the enclave at the target segment.

```
void _gcry_mpi_ec_mul_point ( mpi_point_t result,
 gcry_mpi_t scalar, mpi_point_t point, mpi_ec_t ctx) {
 /* ... */
 if (mpi_is_secure (scalar)) {
  /* Oblivious Implementation */
 } else {
  /* Implementations with side-channels */
  for (j=nbits-1; j >= 0; j--) {
   _gcry_mpi_ec_dup_point (result, result, ctx);
   if (mpi_test_bit (scalar, j))
    _gcry_mpi_ec_add_points (result, result, point,
                     ctx);
  }
 }
}
```

[illegible] `mpi_ec_mul_point` [illegible]

Further, we target `mpi_ec_dup_point` when launching the Prime-and-Probe attack.

5 [illegible] LIBGCRYPT

We instantiated the relevant routines of `libgcrypt 1.7.0` in SGX, by using Panoply [28]. Our experiments were carried out with Ubuntu 18.04 on an Intel E3-1280 with 4 physical cores and 32GB RAM.

To assess the effectiveness of our attack strategy in evading existing detection tools, we compiled and run the two cryptographic routines using T-SGX with some engineering efforts. Since the other two detection tools available in literature are not released as open-source—Varys is part of a commercial product and Déjà Vu is no longer maintained—we also evaluate the effectiveness of our strategy on enclaves equipped with a comprehensive tool, dubbed Monitor++, assumed to monitor all of the performance metrics proposed in literature (i.e., number of AEX, cache misses, and execution time). Monitor++ raises an alarm if the witnessed performance is anomalous. We note that cache misses are typically monitored via performance counters—a feature that is not currently available for SGX enclaves. Nevertheless, previous work has shown that non-SGX applications could use cache-misses to detect cache-based attacks [10]; hence, we also include the number of cache misses among the performance metrics that are monitored by Monitor++, to capture the possibility that it becomes available to future SGX applications.

5 [illegible]

We start by measuring the execution time of one loop of the victim routines—recall that a loop of `mpi_powm` and `mpi_ec_mul_point` is a code segment as defined in Section 3. We do so by running each loop 100 times with secret bit `0` and another 100 times with secret bit 1.

Our results show that, in case of using Monitor++, a "`0`-loop" of `mpi_powm` takes on average 46.4k clock ticks (σ = 493.6), while a "1-loop" takes on average 92.9k clock ticks (σ = 122.2). When instrumented with T-SGX, `mpi_powm` takes slightly longer: 48.3k clock ticks (σ = 530.1) for a `0`-loop and 103.2k clock ticks (σ = 251.3) for a 1-loop.

Routine `mpi_ec_mul_point` with Monitor++, takes on average 15.4k clock ticks (σ = 378.1) for a `0`-loop, and 39.2k clock ticks (σ = 284.9) for a 1-loop. When instrumented with T-SGX, the function `mpi_ec_ec_mul_point` nearly double its computation time: it takes 38.8k clock ticks (σ = 631.3) and 92.0k clock ticks (σ = 376.1) for `0`-loop and 1-loop, respectively.

	[illegible]
`mpi_powm` (w/ Monitor++)	93.17% (σ = 5.49%)
`mpi_powm` (w/ T-SGX)	77.68% (σ = 10.90%)
`mpi_ec_mul_point` (w/ Monitor++)	81.74% (σ = 4.52%)
`mpi_ec_mul_point` (w/ T-SGX)	67.33% (σ = 3.40%)

[illegible]

Once we have the running times for `0`-loop and 1-loop iterations, we validate the accuracy of our profiling technique by checking whether we can stop the enclave at the start of each loop. To do so, we fix a random 256 bit secret and we execute the enclave 256 times, each time stopping it at time $T_i - c$ (with i = 1,...,256 and c = 5,000 clock ticks).[1] In order to learn the ground truth, we inject a counter into the code to keep track of the number of loop iterations thus far. This experiment is repeated 20 times and we report the results in Table 1.

Stopping a victim enclave equipped with Monitor++ at the start of a loop, leverages the fact that the victim exposes page fault to the OS. However, if the victim uses T-SGX, we note that page-faults are dispatched to the enclave abort-handler so that the OS is not notified. We therefore leverage a technique similar to Prime+Abort [15]. In particular, we leverage TSX and setup a transaction with a cache set that conflicts with the memory address that starts the execution of a loop at the victim. As a result, our transaction aborts as soon as the victim starts a loop.

Our evaluation shows that stopping at a specific code segment an enclave running `mpi_powm` with Monitor++ is more accurate (93.17%) than achieving the same if the enclave runs the routine instrumented with T-SGX (77.68%). This is because we may lose synchrony with the victim as T-SGX restarts a transaction. We observe the same behavior for `mpi_ec_mul_point`: 81.74% for the version using Monitor++ and 67.33% for the version instrumented with T-SGX. A comparison between `mpi_powm` and `mpi_ec_mul_point` shows lower accuracy for the latter. This is because one loop of `mpi_ec_mul_point` takes less time to complete compared to a loop of `mpi_powm`—therefore, it is harder to hit the start of a specific loop iteration.

5 [illegible]

We evaluate the accuracy of our attack variants in recovering secret bits when the victim is either equipped with Monitor++ or with T-SGX. In case of T-SGX, we do not use attack variants that leverage cache since the original T-SGX paper does not address cache-based attacks [26].

We fix a random 256 bit secret and, for i = 1,...,256, we recover the secret bit s_i by attacking the corresponding code segment 9 times (i.e., for 9 times we run the enclave and launch the side-channel attack from T_i until T_{i+1}). Given the 9 samples, we determine the secret bit based on majority voting. We repeat the experiment 10 times and show the average accuracy and standard deviation in the column "Attack Accuracy" in Table 2 and Table 3 for `mpi_powm` and `mpi_ec_mul_point`, respectively. For comparison purposes, we also report the accuracy of "standard" side-channel attacks using either

[1] Note that we can correctly estimate any T_i because we know the value of the secret bits.

		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Our attacks	Our-PF	Monitor++	85.5% (σ = 7.9%)	3.04 (σ = 0.20)	125.91 (σ = 82.72)	5.67 (σ = 0.031)
	Our-PFCa		69.8% (σ = 6.1%)	2.70 (σ = 0.53)	175.05 (σ = 135.05)	5.64 (σ = 0.014)
	Our-Ca (w = 3)		76.3% (σ = 10.1%)	1.32 (σ = 0.46)	164.05 (σ = 47.06)	5.62 (σ = 0.010)
	Our-Ca (w = 5)		89.14% (σ = 13.54%)	1.39 (σ = 0.48)	218.81 (σ = 23.67)	5.62 (σ = 0.0099)
	Our-Ca (w = 9)		99.7% (σ = 0.5%)	1.30 (σ = 0.46)	275.76 (σ = 38.02)	5.62 (σ = 0.011)
	Our-Prime+Abort	T-SGX	71.03% (σ = 3.17%)	6.82 (σ = 10.38)	431.76 (σ = 415.22)	5.71 (σ = 0.04)
Standard attacks	Page-faults attack	Monitor++	97.9% (σ = 3.2%)	831.97 (σ = 99.69)	124.16 (σ = 21.78)	11.50 (σ = 0.758)
	Cache attack		89.5% (σ = 4.3%)	1.67 (σ = 0.71)	2112.86 (σ = 82.10)	5.68 (σ = 0.0078)
	Prime+Abort	T-SGX	88.1% (σ =1.1%)	577.22 (σ = 99.43)	7834.45 (σ = 1984.30)	5.84 (σ = 0.09)
No attack	`mpi_powm`	Monitor++		2.441 (σ = 1.93)	123.27 (σ = 82.91)	5.63 (σ = 0.017)
	`mpi_powm` (w/ GCC)			14.44 (σ = 10.97)	495.0 (σ = 290.11)	5.78 (σ = 0.09)
	`mpi_powm` (w/ Redis)			1.48 (σ = 0.70)	6007.21 (σ = 510.83)	6.05 (σ = 0.48)
	`mpi_powm`	T-SGX		6.10 (σ = 21.76)	416.48 (σ = 410.76)	5.66 (σ = 0.02)
	`mpi_powm` (w/ GCC)			188.67 (σ = 85.14)	394.14 (σ = 549.7)	5.71 (σ = 0.60)
	`mpi_powm` (w/ Redis)			121.25 (σ = 120.68)	16256.47 (σ = 5843.78)	6.12 (σ = 0.25)

[illegible] **`mpi_powm`.**

		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Our attacks	Our-PF	Monitor++	69.6% (σ=3.3%)	3.01 (σ = 0.12)	98.16 (σ = 14.12)	6.31 (σ = 0.012)
	Our-PFCa		64.4% (σ=2.7%)	2.33 (σ = 0.47)	155.94 (σ = 112.04)	6.30 (σ = 0.010)
	Our-Ca (w = 3)		86.4% (σ=12.91%)	1.60 (σ = 0.49)	186.43 (σ = 16.99)	6.29 (σ = 0.012)
	Our-Ca (w = 5)		100%	1.50 (σ = 0.49)	201.41 (σ = 22.22)	6.29 (σ = 0.011)
	Our-Ca (w = 9)		100%	1.50 (σ = 0.50)	249.38 (σ = 22.75)	6.29 (σ = 0.012)
	Our-Prime+Abort	T-SGX	70.1% (σ=2.5%)	7.04 (σ = 9.25)	219.07 (σ = 116.8)	14.02 (σ = 0.48)
Standard attacks	Page-faults attack	Monitor++	99.6% (σ=0.22%)	499.28 (σ = 96.31)	98.80 (σ = 21.09)	10.19 (σ = 0.76)
	Cache attack		96.8% (σ=5.0%)	1.47 (σ = 0.50)	9684.87 (σ = 701.08)	6.46 (σ = 0.011)
	Prime+Abort	T-SGX	98.9% (σ=1.8%)	695.89 (σ = 72.91)	18605.17 (σ = 2646.15)	13.53 (σ = 0.11)
No attacks	`mpi_ec_mul_points`	Monitor++		2.71 (σ = 2.28)	106.23 (σ = 60.79)	6.29 (σ = 0.012)
	`mpi_ec_mul_points` (w/ GCC)			23.21 (σ = 27.83)	1246.31 (σ = 1331.89)	6.30 (σ = 0.02)
	`mpi_ec_mul_points` (w/ Redis)			1.61 (σ = 0.83)	6092.45 (σ = 1043.90)	7.21 (σ = 1.02)
	`mpi_ec_mul_points`	T-SGX		6.22 (σ = 34.48)	158.93 (σ = 133.85)	13.31 (σ = 0.29)
	`mpi_ec_mul_points` (w/ GCC)			377.93 (σ = 87.54)	817.54 (σ = 1331.89)	13.67 (σ = 0.50)
	`mpi_ec_mul_points` (w/ Redis)			226.91 (σ = 109.03)	84941.38 (σ = 25664.85)	17.69 (σ = 1.17)

[illegible] **`mpi_ec_mul_point`.**

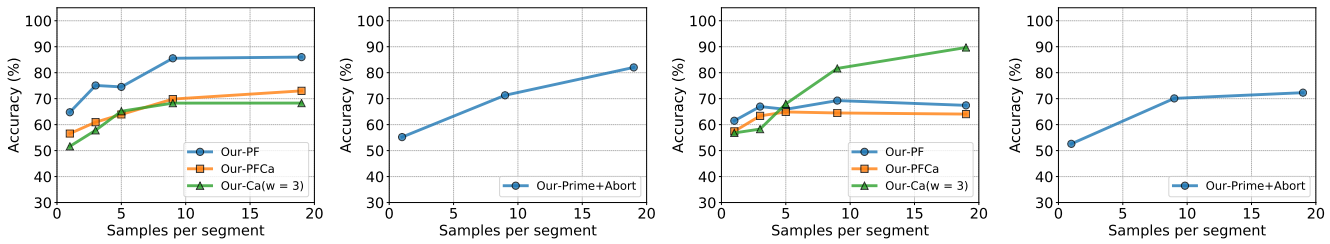
page-faults [32] or L3 cache [18, 32]. A standard attack refers to an attack that runs throughout the whole execution of the victim in order to recover the largest number of secrets bits in one execution. In case of standard attacks we also repeat the attack 9 times and use majority voting to decide the value of each secret bit. Note that in case of routines instrumented with T-SGX, a standard page-faults attack does not work as T-SGX does not expose page-faults to the OS. In this case, we use the Prime+Abort attack of [15].

Column "Attack Accuracy" of Table 2 and Table 3 show that our attack strategy can recover between 70% and 100% of the enclave secret, depending on (i) the type of side-channel exploited, (ii) the detection tool used by the victim (either T-SGX or Monitor++), and (iii) the number of consecutive code segments attacked per run for attacks that only exploit cache. Our experiments also show that attack accuracy decreases when the victim is instrumented with T-SGX: this is likely due to the noise introduced by T-SGX when restarting transactions that abort before completion.

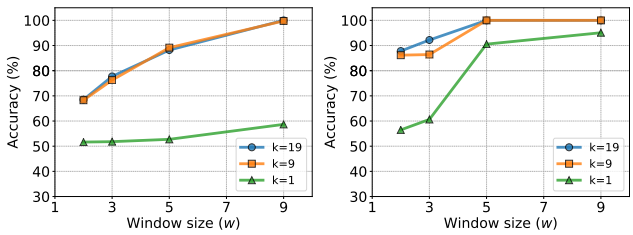
Comparing the accuracy of attacks on `mpi_powm` with the accuracy when attacking `mpi_ec_mul_point`, we note that Our-PF performs better on `mpi_powm` and this is because the time difference between a `1`-loop and a `0`-loop in that routine is sharper than the time difference of the loops in `mpi_ec_mul_point`. Nevertheless, attack variants that use cache are more accurate on `mpi_ec_mul_point` as the cache side-channel is more noisy when attacking `mpi_powm`. Furthermore, cache-only attack with larger windows (e.g., $w = 9$) provide very good results.

We also assess the impact on accuracy of the number of samples k we obtain for each secret bit. As shown in Figure 9, increasing k improves accuracy that, however, plateaus around $k = 9$ for most of the attack variants.

Finally, we assess the impact on accuracy when relying on a sliding window to reduce alignment errors in cache-only attacks. Recall that attacking a single segment at a time by only using cache side-channels may lead to poor results due to the difficulty of aligning the victim and attack threads (see Section 3). In our experiments, a cache-only attack on one segment at a time resulted in an average accuracy over 20 runs of 46.64% (σ = 3.84%) for `mpi_powm` with Monitor++. The same experiment when attacking `mpi_ec_mul_point` showed an average accuracy of 51.64% (σ = 1.98%). By using the sliding window technique described in Section 3, we improve accuracy as shown in Figure 10. In particular, a window of size $w = 9$ allows to recover the full secret when attacking `mpi_powm`, whereas the same result can be achieved with a window of size $w = 5$ for `mpi_ec_mul_point`. This is



(a) `mpi_powm` **(b)** `mpi_powm` **(c)** `mpi_ec_mul_point` **(d)** `mpi_ec_mul_point`



(a) `mpi_powm`. **(b)** `mpi_ec_mul_point`.

because, the cache side-channel is less noisy in `mpi_ec_mul_point`, as explained before.

5

We now assess the effectiveness of T-SGX and Monitor++ described above in detecting an adversary using any of our attack strategies. Recall that T-SGX monitors the number of transaction aborts whereas Monitor++ monitors performance metrics proposed in literature, namely number of AEXs, execution time, and cache misses.

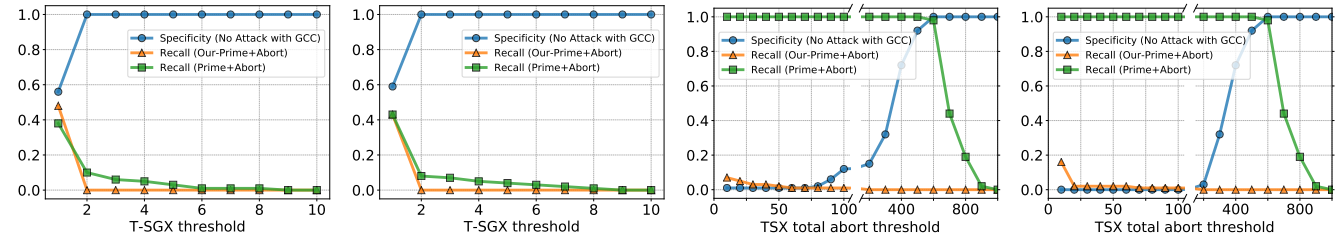
We collect the aforementioned performance metrics, when the victim is under attack, as well as when the victim is running in a benign environment either (i) alone or (ii) while another process is running on the same machine. For the latter, we used either `Redis`—a key-value store—and we mimic a realistic workload as in [4], or GCC while building a large project. Finally, we record the required performance metrics while attacking the victim with standard side-channel attacks.

On the one hand, reported figures on routines instrumented with T-SGX provide us with evidence of the effectiveness of our attack strategies when the victim is equipped with existing tools. On the other hand, results of the experiments with Monitor++ allow us to reason about effectiveness of our attack strategies with respect to any tool that monitors the performance of the potential victim. For each scenario, we run the victim 1,000 times and report the average and standard deviation of each metric in Table 2 and Table 3. In particular, the tables show the number of AEX for Monitor++ and the total number of TSX aborts across transactions for T-SGX.

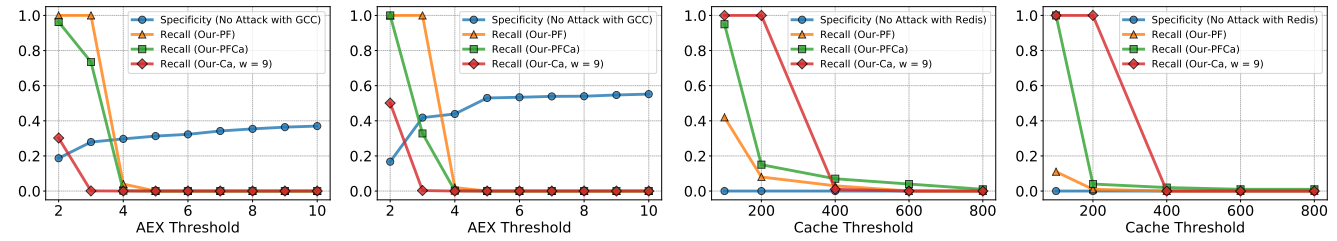
For each of the considered scenarios and for different values of the detection thresholds, we also measure *recall* (ratio of true positives over all positive cases) and *specificity* (ratio of true negatives over all negative cases). The recall metric is used to measure security: a perfect tool should have recall equal to one, and any smaller value means that the tool is not detecting some attacks. Specificity measures usability: a perfect tool should have specificity equal to one, and any smaller value means that the tool is raising some false alarms. In a real deployment, detection thresholds should be set so that both recall and specificity are as close as possible to one. In the following, we show that, for both T-SGX and Monitor++, high specificity and high recall cannot be achieved at the same time—which, in turn, shows that such tools cannot detect an adversary that uses our attack strategy.

In this set of experiments, we run the victim instrumented with T-SGX along with GCC to mimic a realistic multi-threaded workload. We vary the threshold number of transaction aborts before T-SGX raises an alarm, and measure specificity and recall for each threshold.

Figure 11 (a) and (b) depicts the results for `mpi_powm` and `mpi_ec_mul_point`. In order to reach a decent level of specificity (i.e., to avoid false-alarms), one should set the detection threshold $t \geq 2$. This result is inline with the experiments of the original T-SGX paper [26] that reports that most of the transactions of the applications abort and must be restarted up to two times before completing. However, if $t \geq 2$, all of our attacks go undetected (i.e., recall is 0). For completeness, Figure 11 (a) and (b) also reports the recall value for the "standard" Prime+Abort attack of [15]: it is roughly 0.1 for $t = 2$ and decreases to 0 when $t \geq 8$. In other words, T-SGX cannot detect Prime+Abort attacks. We note however, that Prime+Abort can be easily detected by T-SGX if it monitored the total number of transaction aborts, apart from the number of aborts per transaction. To show this, we have modified T-SGX to keep track of the number of aborts across all transactions and to raise an alarm if that reaches a specified threshold t'. Figure 11 (c) and (d) shows specificity and recall for `mpi_powm` and `mpi_ec_mul_point`. It takes a threshold t' of roughly 600 aborts to reach specificity close to one, in order to avoid false positives. Nevertheless, if $t' = 600$, the standard TSX-based attack can be easily detected whereas Our-Prime+Abort goes unnoticed for $t' \geq 100$. Further, a Prime+Abort attack could be easily spotted by monitoring the victim's cache. In our experiments, Prime+Abort against `mpi_powm` caused, on average a ×10 increase of cache misses, compared to a benign execution. The increase of cache misses if the victim were running `mpi_ec_mul_point` was ×19 on average.



(a) mpi_powm (b) mpi_ec_mul_points (c) mpi_powm (d) mpi_ec_mul_points



(a) mpi_powm (b) mpi_ec_mul_points (c) mpi_powm (d) mpi_ec_mul_point

In case of Monitor++, we consider detection based on both the number of AEXs and the number of cache misses. For each scenario, we run the victim either along with GCC—to mimic a multi-threaded workload—or along with Redis—as an exemplary application to mimic a memory-intensive workload. Further, we vary the detection threshold—either the one of number of AEXs or the one of number of cache misses— and measure specificity and recall for each threshold.

Figure 12a and Figure 12b show results when the tool is monitoring AEXs and the victims are `mpi_powm` or `mpi_ec_mul_point`, respectively. No threshold in the range we consider ($2 \leq t \leq 10$) provides specificity greater than 0.37 in case of `mpi_powm` and 0.55 in case of `mpi_ec_mul_point`. At the same time, all of the attack variants go undetected (recall is 0) if $t \geq 5$ for both victims.

Figure 12c and Figure 12d depict our results when the tool is monitoring cache misses and the victims are `mpi_powm` or `mpi_ec_mul_point`, respectively. For both victims, specificity is 0 for thresholds up to 800; hence, one should set a much higher threshold to avoid false alarms. However, all of the attacks go undetected if $t \geq 800$ for `mpi_powm` or $t \geq 600$ for `mpi_ec_mul_point`.

A detection tool may monitor the execution time to decide whether the application is under attack. This is for example the case of Déjà Vu [14]. Results from Table 2 and Table 3 show that standard page-fault attacks almost double the execution time and would be likely detected by tools such as Déjà Vu. Differently, our attacks cause minimal increase of the victim's execution time (below 2%); as such, it is challenging for Déjà Vu or similar tools to detect them.

5

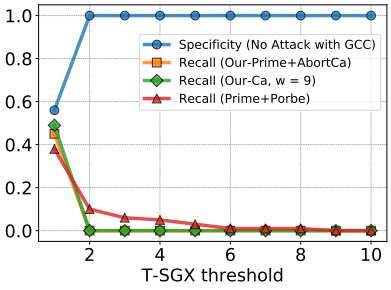
So far, we have not considered cache-based attacks against T-SGX. This is because T-SGX [26] does not monitor the cache performance of an enclave and considers cache-based attacks as out of scope. In this section, we show that even if T-SGX were to be enhanced with additional functionality from Monitor++ (such as monitoring cache performance), it would still not be able to detect our fine-grained attacks.

If T-SGX were to be enhanced with the performance monitoring of Monitor++, the result is a detection mechanism that (i) uses TSX to suppress page-faults notification to the OS, (ii) keeps track of the number of aborts per transaction, and (iii) monitors the number of cache misses. We then use cache-based attacks against this enhanced version of T-SGX and assess whether it can distinguish attacks from benign runs.

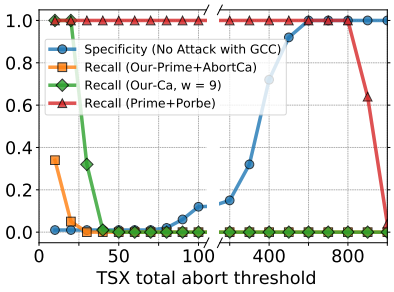
Table 4 and Table 5 summarize our findings for `mpi_powm` and `mpi_ec_mul_point`. As expected, Our-Prime+AbortCa performs slightly worse than Our-PFCa (e.g., 70% vs 60% accuracy for `mpi_powm`): this is because accuracy of stopping at a specific code segment an enclave is higher if the enclave exposes page-faults. We also note that attack strategies that only leverage cache can recover the whole secrets with alignment windows of size $w = 9$.

Regarding performance metric, we note that a standard cache attack noticeably increases the number of cache misses at the victim compared to benign run when, for example, GCC is running on the same host (between ×10 and ×20) whereas Our-Ca causes a number of cache misses at the victim that is comparable to a benign run with no other application on the same platform.

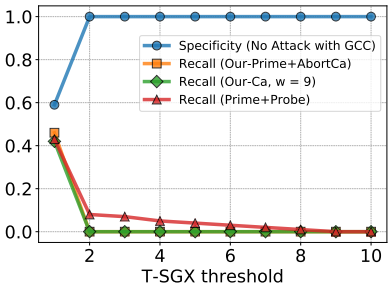
Finally, Figure 13 and Figure 14 show precision and recall of T-SGX and the enhanced T-SGX detection tool against cache-based attacks, respectively. Figure 13 shows that the threshold of allowed aborts per transaction should be at least 2, in order to have specificity



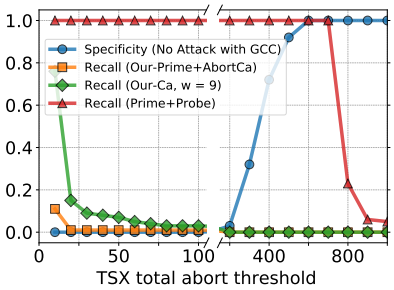
(a) mpi_powm



(a) mpi_powm



(b) mpi_ec_mul_points



(b) mpi_ec_mul_points

close to 1 (so to avoid false positives). In this case, all cache-based approaches are likely to go undetected (recall close to 0); we note that a standard "Prime+Probe" attack (recall around 0.1) performs slightly worse than our attacks (recall close to 0).

If T-SGX were modified to also monitor the total number of transaction aborts, Figure 14 suggests that a threshold of at least 600 must be used to achieve precision close to 1 and avoid false positives. With this threshold, a standard "Prime+Probe" attack can be detected (recall is 1) while our cache-based attacks remain unnoticed (recall is 0).

6

We now adapt our attack strategy to known side-channels of decision-tree routines [22] to assess the feasibility of attacking the decision-tree routine of OpenCV [3]—a well-known computer vision library. Similar to previous work [1], we use the MNIST [2] data-set and assume an application consisting of an enclaved execution of OpenCV's decision-trees to detect handwritten digits.

The decision-tree traversal function of OpenCV walks the tree and, depending on the input image, accesses different nodes, resulting in different page accesses. We use page-faults to infer the pattern of page accesses and leak the prediction output. To capture the access pattern of different input images, we rely on an offline analysis of the routine and observe memory page access patterns. Thus, we set such memory pages as fault during runtime, to infer the prediction output.

For training the decision-tree, we rely on 60,000 samples from the MNIST data set. During the inference phase, the trained decision-tree model is first loaded into the enclave and then used to recognize 100 input images at a time from a set of 10,000 test images. The execution time of image recognition is almost independent of the input image, as the tree is almost balanced. Therefore, we can model the execution time as $T_i = T_i + c$, where c is a constant value. In our experiments, we found out that c is roughly 9.7k clock ticks ($\sigma = 975.3$).

We successfully stopped the enclave at the i-th invocation of `DTreesImpl::predictTrees` around 8 out of 10 times (84.8% ($\sigma = 7.85\%$)). The corresponding accuracy is reported in Table 6. We observe that a variant attack leveraging page-faults (Our-PF) is only slightly less accurate than a standard page-fault attack.

To analyze the effectiveness of our strategy against detection tools, we measure specificity and recall for different AEX thresholds. We only assume the victim is equipped with Monitor++—as T-SGX supports only C, we could not instrument OpenCV using T-SGX. Figure 15 shows that no threshold value can achieve high specificity and high recall at the same time. In particular, if the detection threshold is smaller than 17, then specificity falls below 0.84 (i.e., a false alarm is raised 2 out of 10 times). At the same time, a detection threshold equal to or bigger than 11 allows attacks to go undetected (recall=0.003). We also note that Our-PF causes no noticeable overhead in terms of cache misses or execution time. We conclude that Monitor++—monitoring number of AEXs, cache misses or execution time—may not be able to tell an attack that uses our strategy from a benign run of the victim enclave.

Our attacks	Our-Prime+AbortCa	60.2% (σ=3.4%)	9.56 (σ=3.85)	247.69 (σ=153.40)	5.71 (σ=0.02)
	Our-Ca (w=3)	76.10% (σ=10.52%)	7.11 (σ=18.46)	237.06 (σ=115.11)	5.66 (σ=0.06)
	Our-Ca (w=5)	86.42% (σ=14.32%)	18.59 (σ=1.01)	284.94 (σ=69.83)	5.68 (σ=0.05)
	Our-Ca (w=9)	100%	28.16 (σ=3.74)	388.20 (σ=87.73)	5.67 (σ=0.04)
Standard attacks	Cache attack	84.4% (σ=9.6%)	923.18 (σ=55.52)	3697.68 (σ=817.95)	5.81 (σ=0.013)
No attack	`mpi_powm`		6.10 (σ=21.76)	416.48 (σ=410.76)	5.66 (σ=0.02)
	`mpi_powm` (w/ GCC)		188.67 (σ=85.14)	394.14 (σ=549.7)	5.71 (σ=0.60)
	`mpi_powm` (w/ Redis)		121.25 (σ=120.68)	16256.47 (σ=5843.78)	6.12 (σ=0.25)

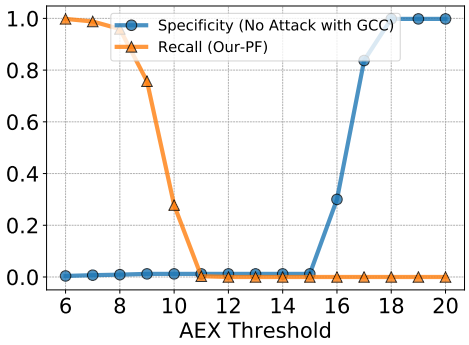
`mpi_powm`.

Our attacks	Our-Prime+AbortCa	55.3% (σ=2.5%)	6.06 (σ=15.54)	303.66 (σ=344.21)	13.95 (σ=0.40)
	Our-Ca (w=3)	61.44% (σ=14.1%)	17.65 (σ=42.48)	268.35 (σ=562.07)	13.45 (σ=0.26)
	Our-Ca (w=5)	75.91% (σ=10.1%)	18.73 (σ=43.10)	257.92 (σ=448.48)	13.70 (σ=0.32)
	Our-Ca (w=9)	100%	23.9 (σ=55.37)	305.34 (σ=524.03)	13.36 (σ=0.24)
Standard attacks	Cache attack	96.6% (σ=3.8%)	788.89 (σ=67.50)	15730.48 (σ=4363.18)	14.17 (σ=0.19)
No attack	`mpi_ec_mul_points`		6.22 (σ=34.48)	158.93 (σ=133.85)	13.31 (σ=0.29)
	`mpi_ec_mul_points` (w/ GCC)		377.93 (σ=87.54)	817.54 (σ=1331.89)	13.67 (σ=0.50)
	`mpi_ec_mul_points` (w/ Redis)		226.91 (σ=109.03)	84941.38 (σ=25664.85)	17.69 (σ=1.17)

`mpi_ec_mul_point`.

No attack	`DTreesImpl::predictTrees`		7.83 (σ=0.49)	134.55 (σ=85.38)	2.46 (σ=0.06)
	`DTreesImpl::predictTrees` (w/ GCC)		16.74 (σ=1.44)	132.42 (σ=73.18)	2.55 (σ=0.03)
Standard attack	Page-faults attack	65.2% (σ=0)	3070.9 (σ=1.4)	2164 (σ=5455.22)	58.81 (σ=0.41)
Our attack	Our-PF	54.9% (σ=3.61%)	8.21 (σ=1.34)	150.77 (σ=74.67)	2.47 (σ=0.08)

`DTreesImpl::predictTrees`



7

In this paper, we analyzed the limitations of existing detection tools that monitor performance metrics to detect side-channel attacks on SGX enclaves. Our findings show that an adversary can bypass existing tools when deployed in realistic cloud scenarios by exfiltrating small portions of a secret at each run of the victim.

One possible countermeasure would be to instruct detection tools to keep *state* to detect a pattern of small anomalies spread across multiple executions. Intel SGX, however, does not provide freshness of state information sealed to disk. A malicious OS can, therefore, easily bypass such a tool by providing stale state to the enclave. Another countermeasure could be to prevent arbitrary restarts of the victim enclave by, e.g., programming the enclave to run only upon receiving an authenticated request. Nevertheless, this option is hardly workable when the enclave provides a "public" service. For instance, if the enclave hosts a TLS server [6] or a password-hardening service [17], it is extremely challenging to differentiate between an authorized request from a honest user and another issued by the adversary acting as a honest user.

We hope that our findings will motivate further research in this area, with the aim to avoid unnecessary cycles of attacks/defenses on detection tools that solely rely on performance metrics.

We thank all anonymous reviewers for their helpful comments. This work was partially funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957406 (TERMINET), and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

[1] 97% on MNIST with a single decision tree (+ t-sne). https://www.kaggle.com/carlolepelaars/97-on-mnist-with-a-single-decision-tree-t-sne.
[2] Mnist dataset. http://yann.lecun.com/exdb/mnist/.
[3] Opencv. https://github.com/opencv/opencv.
[4] Redis benchmark. https://redis.io/topics/benchmarks.

[5] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A commodity obfuscation engine on intel sgx. In *NDSS*, 2019.

[6] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David M. Eyers, and Peter R. Pietzuch. Libseal: revealing service integrity violations using trusted execution. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys*, pages 24:1–24:15, 2018.

[7] Maurice Bailleu, Donald Dragoti, Pramod Bhatotia, and Christof Fetzer. Tee-perf: A profiler for trusted execution environments. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 414–421. IEEE, 2019.

[8] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. Dr.sgx: Automated and adjustable side-channel protection for sgx using data location randomization. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, pages 788–800, New York, NY, USA, 2019. ACM.

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies (WOOT)*, pages 1–12, 2017.

[10] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In Ziming Zhao, Gail-Joon Ahn, Ram Krishnan, and Gabriel Ghinita, editors, *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, pages 224–235. ACM, 2018.

[11] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, 2017.

[12] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T. Lai, and D. Lin. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 178–194, May 2018.

[13] Sanchuan Chen. Déjà Vu. https://github.com/schuan/dejavu. Accessed on 22/11/2021.

[14] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *ACM Asia Conference on Computer and Communications Security, (AsiaCCS)*, pages 7–18, 2017.

[15] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. Prime+abort: A timer-free high-precision L3 cache attack using intel TSX. In *26th USENIX Security Symposium, USENIX Security*, pages 51–67, 2017.

[16] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, Vancouver, BC, August 2017. USENIX Association.

[17] Arseny Kurnikov, Klaudia Krawiecka, Andrew Paverd, Mohammad Mannan, and N. Asokan. Using safekeeper to protect web passwords. In *The Web Conference, WWW*, pages 159–162, 2018.

[18] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[19] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015.

[20] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 69–90, 2017.

[21] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled instruction-level attacks on enclaves. In *29th USENIX Security Symposium*, pages 469–486, August 2020.

[22] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, 2016.

[23] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *USENIX Annual Technical Conference (ATC)*, pages 227–240, 2018.

[24] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. Cosmix: A compiler-based system for secure memory instrumentation and execution in enclaves. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 555–570, Renton, WA, July 2019. USENIX Association.

[25] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference (DIMVA)*, pages 3–24, 2017.

[26] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Network and Distributed System Security Symposium 2017 (NDSS'17)*, February 2017.

[27] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *ACM Asia Conference on Computer and Communications (AsiaCCS)*, pages 317–328, 2016.

[28] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*, 2017.

[29] SSLab@Gatech. T-SGX. https://github.com/sslab-gatech/t-sgx. Accessed on 22/11/2021.

[30] O Sury and R Edmonds. Edwards-curve digital security algorithm (eddsa) for dnssec. Technical report, RFC 8080 (Proposed Standard). Internet Engineering Task Force, 2017.

[31] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[32] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2421–2434, 2017.

[33] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. *ACM SIGARCH Computer Architecture News*, 45(2):81–93, 2017.

[34] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (SP)*, pages 640–656, 2015.