Aion Attacks: Manipulating Software Timers in Trusted Execution Environment

Wei Huang[1], Shengjie Xu[1], Yueqiang Cheng[2], and David Lie[1]

[1] University of Toronto, Toronto, Canada
{wh.huang,shengjie.xu}@mail.utoronto.ca, lie@eecg.toronto.edu
[2] NIO, San Jose, USA yueqiang.cheng@nio.io

Abstract. Side-channel attacks are a threat to secure software running in a Trusted Execution Environment (TEE). To protect Intel SGX applications from these attacks, researchers have proposed mechanisms to detect cache-probing and repeated interrupts that these attacks rely on. These defenses often rely on high-resolution timers. However, since there is no trusted high-resolution timer hardware module, developers have resorted to software timers, which unfortunately underestimate the scope of possible attacks. In this paper, we propose *Aion* attacks that manipulate the speed of a reference software timer to subvert defensive mechanisms against SGX side-channel attacks. Specifically, we introduce a CPU thermal attack that leverages the thermal management mechanism to change the execution speed of the timer thread, and a cache eviction attack that evicts the target timer counters and forces the system to load them from memory instead of cache. We evaluated the above Aion attacks and introduced an analytical model and show that software timers cannot be improved to fit the defenders under our attacks.

1 Introduction

The threat model of Intel SGX assumes that only CPUs are trustworthy, placing code and data of protected applications in a secure enclave isolated from other system software. However, the protection guaranteed by SGX does not take into account an attacker who monitors information leakage via side-channels. As a result, various defense mechanisms have been proposed [5,17] to defend against side-channel attacks on SGX applications [25,4,19,1,15,10,8,7,3].

Many of these defenses rely on the ability to measure the frequency and duration of certain events, such as cache access and code execution time, or the number of asynchronous enclave exits. Since SGX does not provide a trusted hardware timer, these defenses instead use high-resolution software timers to measure the passage of time. All software timers make assumptions about the processor they are executing on: They assume that 1) CPU instructions execute at a relatively constant speed, and that 2) the clock frequency the CPU operates at stays within a well-defined range. For example, they acknowledge that SGX must defend against an adversary who might modify the processor clock frequency. Thus, they are resilient to an adversary who can slow down the clock

frequency by a factor of $3.25 - 4.25\times$, as this is the typical ratio between the maximum and minimum operating clock frequency of modern processors.

In this paper, we show that both of these assumptions can be broken by a significant margin. We present *Aion* attacks, which can be mounted by both privileged and unprivileged attackers, and enable an adversary to tamper with software timer accuracy by $2.5 - 202\times$. We also build a model of software timers and show empirically that secure software timers are not possible on current architectures. This renders all current software-timer-based SGX side-channel defenses useless.

Our Aion attacks use two mechanisms to break the assumptions made by software timers. The first attack manipulates the thermal management facilities of the processor to cause execution slowdown below that of the lowest supported clock frequency of the processor, violating the assumption that slowdowns are bounded by the lowest clock frequency. As far as we are aware, this is the first instance of a security attack that abuses CPU thermal management, and does not actually need to physically overheat or damage the CPU in any way. Instead, we trigger thermal management features using software-only attacks. The second attack generates cache evictions to slow down the execution of instructions in the software timer, violating the assumption that the execution time of instructions is relatively constant. We show these attacks can compromise the security properties of applications running in SGX enclaves, and can allow existing side-channel attacks to evade detection by existing defenses. We implement a prototype of Aion attacks and evaluate them in a real-world environment. We make the following contributions in this paper:

- We propose an analytical model which suggests that no existing software timers used in SGX enclaves are reliable, meaning that current SGX side-channel defences are ineffective if timers are manipulated by attackers.
- We present two generic Aion attacks and show that they are able to effectively exploit all existing SGX software timers, invalidating current defense mechanisms.
- We evaluate two prototypes of Aion attacks on two different CPUs. Experimental results indicate that both are able to consistently break the desired properties of the software timers. With our mechanism, an end-to-end attack is demonstrated where existing side-channel attacks can evade detection.
- We prove that, under our attack model, it is impossible to build a software timer immune to Aion attacks, motivating the need for hardware support.

The remainder of this paper includes: Section 2 provides related backgrounds. We propose our analytical system model in Section 3 and describe our attack design in Section 4, implementation in Section 5, and evaluations in Section 6. We finally conclude the paper in Section 7.

2 Background and Related Work

2.1 Intel SGX and TSX

Intel Software Guard Extensions (SGX) [11,6] is an instruction set extension introduced in 2015 to the Intel architecture. SGX is designed for security and

system properties such as confidentiality, execution isolation, memory integrity, and verifiability. It provides a trusted execution environment (TEE) for user-level applications to securely run in a shielded environment called an enclave. Security properties are guaranteed by putting application code and data into the processor reserved memory (PRM) which is isolated from the main memory and transparently encrypted by the memory encryption engine (MEE). Secure applications in enclaves can be interrupted by other applications outside the enclave, triggering an asynchronous enclave exit (AEX) event.

Intel Transactional Synchronization Extensions (TSX) [11] is an Intel ISA extension for hardware transactional memory. It ensures that when a sequence of instructions is executed, either the execution is completed without interruption or memory read-write conflicts (i.e., concurrent access to the same data where at least one access is a write) with other threads, or the transaction is aborted and the execution is rolled back. The original purpose of TSX was to speed up multi-threaded applications by reducing locking, while recent work has leveraged it to notify secure enclaves about interruptions by other threads [20] and protect cryptographic keys against memory disclosure attacks [9].

2.2 Power and Thermal Management of Intel CPU

Recent CPUs have power and thermal management features for energy efficiency and protection of the processors from overheating. For example, Intel has several digital thermal sensors (DTS) in each CPU package to monitor processor temperature [11], and the results can be retrieved from model-specific registers (MSRs) or the platform environment control interface (PECI). When a certain temperature limit is reached, a thermal control circuit (TCC) will be activated and it may take following three actions: 1) Reducing core frequencies so the clock runs slower, 2) Reducing the core voltages to make the processors use less power and generate less heat, and 3) Forcing one or more cores to enter a hardware duty cycling (HDC) mode, in which the processor forces its components in the physical package into the idle state for a certain fraction of time. The TCC can be configured by privileged system administrators to automatically activate under certain circumstances. Such thermal events can also be triggered by software, trapping the CPU into a mode where processors reduce their power consumption by clock modulation.

To the best of our knowledge, our attack is the first academic work to use CPU thermal management features for SGX defence exploitation. We point out that if this thermal management feature is maliciously used, CPU execution speed can be a target that is easily manipulated without being detected by threads running on the controlled core.

2.3 Cache-Based Side-Channel Attacks and Defences in SGX

Cache-based side-channel attacks on SGX are based on general timing attacks on cache like Prime+Probe [16] and Flush+Reload [26]. The basic idea is to measure the access times to a series of specific addresses, and use the information to infer whether or not the victims have accessed data in related addresses. With

this knowledge of the victim's memory access patterns, attackers can retrieve confidential data like private keys. General defence mechanisms have also been proposed for detection and prevention of side-channel attacks like CacheD [23], CaSym [2] and CEASER [18].

Cache-based attack methods have been commonly used in exploiting SGX applications. Malware Guard Extension [19] and CacheZoom [15] develop Prime-+Probe type of attacks with the help of a high-resolution timer to distinguish cache hits and misses, and uses the LLC cache channel. Other side-channel attacks [1,8,7,24] are based on similar methods, with the common strategy of using a high-resolution timer to measure the access latency when probing the victim enclave application's cache and infer secret data from the enclave.

Defensive mechanisms against side-channel SGX attacks also depend on high-resolution software timers inside the SGX enclave. Varys [17] defends cache-based side-channel attacks by enforcing that security-sensitive threads be reserved on the same CPU physical cores and detecting attacker threads that attempt to access shared CPU resources. Déjà Vu [5] measures application execution time with a more complex software timer using Intel TSX, but not all of the timer thread is under TSX's protection because some parts of the timer need to be shared across threads. Vulnerabilities in these designs will be further discussed in Sec 3 and 4.

3 System Model

3.1 Model of Software Timers

To establish a software timer model, we first define a concept of *wall time*, denoted T_w, as an imaginary clock that is always accurate and up-to-date with physical time in the real world. We assume that all software timer designs should serve the same purpose: to track the wall time as accurately as possible and provide the current time to software that needs it. Since there is no dedicated hardware available in the enclaves for time, we assume that any software timer would need to use a sequence of instructions to track wall-time and use them to mimic an ordinary clock's behavior. To achieve this, a software timer should maintain a *clock time* T_c, and make it available to other threads that need to learn what the current time is.

In the ideal case, the wall time is proportional to clock time by some constant factor, so the clock time can emulate the wall time by executing a sequence of instructions, and the constant factor is decided by how much time the sequence of instructions take to run. This is the best that a software timer could do since a program inside an SGX enclave has no access to a high-resolution hardware timer that can directly provide wall time. We measure its margin of error from wall-time, and define a measure M_t that indicates how accurate the software timer is comparing with the absolute time:

$$\frac{|\Delta t_c - \Delta t_w|}{\Delta t_w} < M_t \tag{1}$$

In the above form, the clock-time at t_1 and t_2 are T_{c1} and T_{c2}, the wall-times

are T_{w1} and T_{w2}, with the times passing by Δt_c and Δt_w respectively. M_t is generally assumed to be so small in practice that $\Delta t_c \approx \Delta t_w$, and is affected by the execution speed of the sequence of instructions, which is in turn affected by CPU execution speed and memory/cache access speed, so the timer model can take this into account by adjusting the measure M_t.

We can now construct a generalized software timer with the above concepts, and use a global variable V_G to simulate the clock ticks. We call V_G the "clock variable". As different machines have different micro-architectures and speed of operations and accesses to cache, for the same timer algorithm, we use a varying parameter I_c to reflect the relation between variable increase and the clock time. I_c represents how much the clock time increases per tick of V_G in average. With this model, the problem of simulating a dedicated timer is transformed to the problem of using an increasing clock variable to indicate the current time, where the parameter I_c makes it generally adaptable under different settings of different machines.

In this way, when a user or an application needs to measure some Δt_w and since $\Delta t_c \approx \Delta t_w$ (previously assumed due to small M_t), then they can just measure Δt_c to learn how much time it passes in the clock-time and will get the result in the form of:

$$\begin{aligned} \Delta t_c &= T_{c2} - T_{c1} \\ &= I_c \cdot (V_{G2} - V_{G1}) \end{aligned} \tag{2}$$

Where V_{G1} and V_{G2} are the value of V_G at time t_1 and t_2. In this software model, the key problem for system developers is how to determine the value of I_c, and all current software timer approaches assume the value to be relatively stable as it would result in a small accuracy measure M_t. From our previous empirical findings, we have:

$$I_c \propto \frac{T_{Inc(V_G)}}{F_{CPU}} \tag{3}$$

Thus, the value I_c is proportional to $T_{Inc(V_G)}$, the time taken to increment V_G, and reciprocal to F_{CPU}, the CPU speed, which we may estimate to be the average clock frequency of the CPU during the time measured.

To summarize, the software timer model measures the accuracy M_t of a software timer by comparing the clock time it generates with the wall time. The accuracy is affected by an intermediate parameter I_c, which depends on the execution speed of the CPU (F_{CPU}) and the time $T_{Inc(V_G)}$ it needs to increment a global variable. To slow down a software timer, Equation 2 says the adversary should make I_c larger, so that it takes longer to increment the clock variable by some amount. To increase I_c, Equation 3 tells us that the adversary may either make the execution of instructions take longer, or reduce the speed of the CPU.

With these components modeled, we next need to take a look at the victims: The enclave applications and defenders, and how they use the software timers.

3.2 Defender Model

The goal of an SGX side-channel defender is to recognize attacks. A number of defenses do this by measuring the rates or latency of certain events, so they

depend on a software timer. As an example, we here consider two strategies taken by previous work [5,17] that use the rate or the execution time of a measured event as a component of an SGX side-channel defense:

– Cache hit time: The Prime+Probe cache channel attacks on SGX enclaves require the adversary code to run on the same physical core with the victim thread [17], because they have to share the L1/L2 cache to perform the probe. Thus, one method for preventing this attack is filling a core entirely with the application's own threads. To confirm that two threads share the same physical core, we can measure and compare the time that the two threads take to access a shared variable in the L1/L2 cache: if it is an L1/L2 cache hit, the access time should be within around 10 cycles, which implies they share a physical CPU core. A software timer must be used for this measurement.

– Counting Asynchronous Enclave Exits (AEXs): To perform the Prime+Probe cache channel attacks, which give the adversary a fine-grained cache channel for probing, the adversary actively and frequently preempts the target SGX enclave using, for example, the high-precision Advanced Configuration and Power Interface (ACPI) or the High Precision Event Timer (HPET). The preemptions trigger an AEX every time they interrupt the victim enclave, which is an indicator of side-channel attacks if it happens too frequently. The defensive mechanisms can count the number of AEX events during a period of time or measure the time that a certain known sequence of executions takes [17,5] and decide whether the rate of AEX events is too high to raise an alert of side-channel attacks.

In general, these methods all attempt to measure the delay or frequency of some phenomenon. For example, they may monitor whether there is an *irregular* rate of events N_{Ev} (such as AEXs) that happen during a certain period of time. Similarly, the delay of a variable access can be viewed as just the inverse of the number of times the variable is accessed within some period of time. Thus, all tests essentially compare some measured rate of events, N_{Ev}, against a threshold, N_{Th} to detect whether an attack is taking place or not in one of the possible scenarios:

$$N_{Ev} > N_{Th} : attack = true \tag{4}$$

One dilemma that the defender faces is the choice of threshold: Setting N_{Th} too high will result in missed attacks or false negatives, while setting N_{Th} too low will result in false alarms or false positives. The usual solution is to set it according to a calibration run, during which the system is assumed to not be under the influence of an attacker.

However, even then, the defense mechanism must still account for the fact that the measured rate of events N_{Ev} is dependent on both the true rate of events and the ratio between the wall clock and the rate of increment of the clock variable:

$$N_{Ev} = \frac{\Delta t_w}{\Delta t_c} \cdot R_{Ev} \tag{5}$$

where R_{Ev} is the true rate of events according to the wall clock. Normally, we expect that $\Delta t_w \approx \Delta t_c$, so $N_{Ev} \approx R_{Ev}$. However, recall that Δt_c is proportional to I_c in Equation 3. Even under benign conditions, there is some variability to I_c, which may result in some number of false positives and false negatives—typically N_{Th} is set slightly higher so as to bias the detection method for fewer

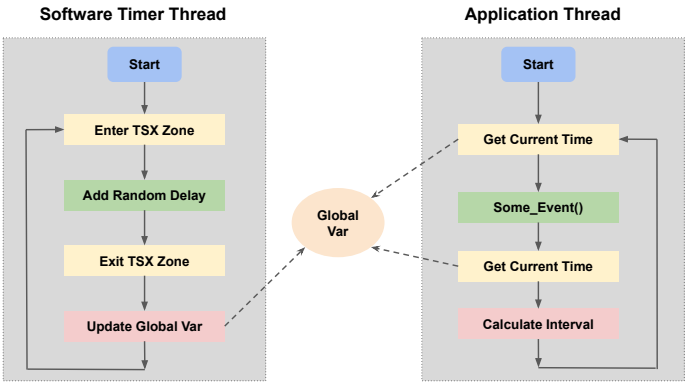

Fig. 1: *A model of how a software timer works and is used by an application thread.*

false positives. However, so long as I_c is similar to the value of I_c during calibration when N_{Th} is set, this will constrain R_{Ev} to be roughly N_{Ev}. Since R_{Ev} corresponds to how fast an attacker is able to read a side-channel, constraining R_{Ev} effectively slows the rate of information leakage to the attacker. However, if the attacker is able to arbitrarily increase I_c, then she can also arbitrarily increase the true rate of events R_{Ev} without being detected. This allows her to probe the side-channel faster and thus reduces the time taken for the attack to extract sensitive information from the enclave.

With this general model of a software timer thread, we now discuss why a best-effort software timer design is still vulnerable to attacker manipulation.

3.3 Timer Countermeasures

We illustrate the general structure of a software timer in Figure 1. A software timer thread updates a global clock variable, which is read by application threads to the current time.

An attacker that wants to tamper with the timer might attempt to interrupt the timer so as to make the difference between Δt_c and Δt_w arbitrarily large. To defend against this, both Déjà Vu and Varys use TSX to detect if the timer has been interrupted. However, TSX can only protect the component of the loop that generates the delay Δt_c, and can not protect the update of the global clock variable, as the clock variable is simultaneously accessed by both the timer thread and application threads. As a result, the update of the global clock variable must be outside the TSX-protected region.

To prevent an attacker from interrupting and delaying the thread as it updates the global variable, TSX is combined with a randomized delay function inside the TSX region, and the global clock variable is updated with the randomized delay. This makes it hard for an attacker to guess when the timer thread is outside of the TSX region and can be interrupted without detection. Thus,

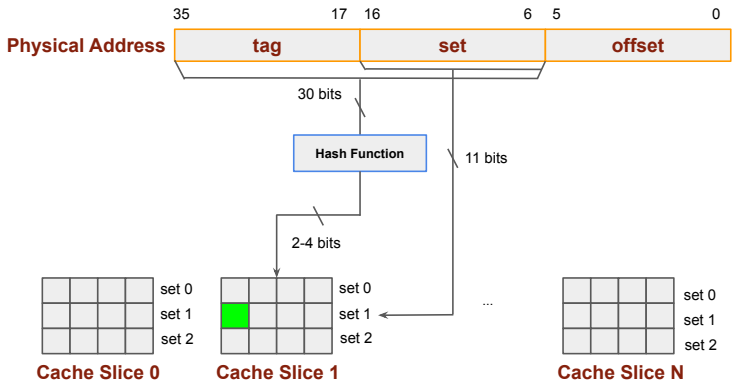

Fig. 2: *Illustration of Intel cache slice and cache set structure.*

we summarize that a secure timer that provides a timing service to other secure application threads needs to include at least the following parts:

1. A global clock variable V_G inside the secure enclave that records the current clock time. The clock time can be read from the clock variable by other threads in the same enclave.

2. A timer loop that increments the clock variable by an amount assumed to be proportional to the amount of time that has passed.

3. A protection mechanism that can either prevent the timer loop from being interrupted or detect if the loop has been interrupted. An example of such a mechanism is TSX.

4. If the entire loop cannot be protected from interruption, a random delay element such that the attacker cannot predict when the timer is in the unprotected region of execution, i.e., right before the clock variable is incremented.

As we can see, a TSX-protected timer should ideally spend a minimal amount of time outside the TSX region. In other words, the only action taken outside of the TSX region should be to increment the clock variable.

With the extra protection of the software timer loop, trivial attacks that manipulate the software timer by interrupting it frequently and/or de-prioritizing the timer thread in an OS thread scheduler to make the software timer deviate from the wall clock would not work. Such scheduler attacks that have OS-level privilege would attempt to preempt the timer thread, which requires interrupting into it. This interruption would be easily caught by the TSX mechanism and the interrupted transaction would be aborted and detected by the defender.

4 Attack Design

4.1 Aion-1: CPU Thermal and Frequency Attack

The *Aion-1* attack manipulates the rate of increase of the clock variable indicating the internal time in the software timer thread, i.e., the F_{CPU} in Equation 3 of our software timer model in Sec 3.1. Intuitively, this can be done by changing the CPU working frequency via the CPU power management modules of the operating system kernel, such that the clock variable increases out of sync with wall time. The strawman method of simply changing CPU frequencies has been described [5], where $procfs$ is used to control the CPU frequency from userspace in on-demand mode, and its effect was generally considered to be bounded by CPU frequency scaling. Taking the Intel i7-6600U as an example: The processor base frequency (PBF) is 2.6GHz; the max turbo frequency (MTF) is 3.4GHz. If the attacker obtains control of a CPU power management module, the minimum controllable frequency of a single core is 800MHz. Thus, it was generally believed that the maximum achievable scale-down of CPU frequency $\Delta t_w/\Delta t_c$ was between 3.25× and 4.25×, a value that most previous defenses could tolerate and still prevent an adversary from mounting an effective attack.

However, our attack can break the above assumptions using CPU thermal management features. As mentioned in Sec 2, Intel CPU thermal management is controlled by a thermal control circuit, whose settings are controlled by a software adaptive thermal monitor. We find that an attacker with root privileges can trigger a thermal event on the CPU thermal control circuit using only software. This not only causes the CPU core frequency and voltage to be throttled down, but can also force the processor into the HDC mode where CPUs are paused for part of the clock duty cycle. Thus, while the clock frequency does not change in HDC mode, the effective execution speed of the CPU is lowered below that of the minimum clock speed, as the CPU is effectively idle for a fraction of the clock cycles. By doing this, we can make the effective execution speed of a CPU approximately equivalent to that of a 100MHz CPU.

According to our software timer model in Sec 3, the accuracy of a software timer depends on the execution speed of the CPU core that the software timer thread runs on. In side-channel defensive mechanisms, the defenders need to make sure the secure enclave has occupied both hyper-threads on the same physical core, such that they do not share L1/L2 cache with other, potentially malicious threads. They do this by measuring the access latency of a shared variable to see if it hits the L1/L2 cache, since if both threads can hit the cache of the same clock variable within around 10 cycles, they must share the same physical core. To evade detection, the Aion-1 attacker only needs to slow down the software timer to make the tester believe that the cache hit time is within 10 cycles in its calibration run, even if it actually hits LLC and takes around 40 cycles or more. It can also do the reverse, depending on which thread the attacker wants to slow down. In this way, any secure application that uses the software timer will read values inconsistent with wall time. This tricks the defender into thinking that the variable access has hit in the L1 cache when it could have hit in the L2 or higher.

The effectiveness of this attack depends on the highest and lowest possible processor execution speed on a single CPU core. The processor execution speed can be regarded as equivalent to the average core frequency during a period of time. When the attack is being mounted, the core that the software timer thread runs on should be set to the lowest possible running speed, and other threads, including the attacker threads, should be set to the highest running speed (or the other way around, when needed). Without the ability to know its own clock speed reliably, the software timer can unknowingly run slower or faster than the original settings. This type of attack also has some limitations, including:
– The attack can only happen if the attackers are able to access CPU MSRs, requiring kernel privileges. In SGX applications on a multi-tenant cloud scenarios, an attacker may not be able to get such privileges as they would need to compromise the hypervisor.
– The attack also assumes the CPU should support thermal control features including clock modulation via MSRs to issue a software signal that activates the TCC. Most of the Intel CPUs available on the market support these features, but not all of them do.

Due to these limitations of privilege and feature availability, we present another attack that uses cache eviction to achieve the goal of manipulating the software timer, possibly as an unprivileged attacker.

4.2 Aion-2: Cache Eviction Attack

The *Aion-2* attack directly targets the clock variable used in the secure software timer thread using an attacker thread in user space. We call this a cache eviction attack, as it slows down the speed of the reference software clock by evicting the clock variable from the CPU cache to DRAM. According to the software timer model in Sec 3, this attack exploits the stability assumption of the cache/memory access speed of the clock variable, i.e., the $T_{Inc(V_G)}$ in Equation 3.

Intel L1/L2 caches are shared by two logical threads on the same physical core, and all threads share the LLC. Because most Intel CPUs use an inclusive cache policy between different levels of cache, evicting the cache line containing the clock variable from LLC would also evict it from L1 and L2 cache. In this way, whenever the software timer thread needs to increment the clock variable, the thread has to wait for extra cycles to complete the request because it has to be served from DRAM. From our experimental results, it takes almost the same number of clock cycles (though not the same wall time) for accessing the same level of cache, and the DRAM access time is about 150 cycles on average, which means the attacker knows how much she can slow down the increment of the clock variable by each eviction. For the rest, the only job that the attacker threads in the user space need to do is to evict the cache line where the clock variable is located. Note that since the clock variable indicating the internal time in the software timer thread is not protected by TSX transactions, access to the clock variable does not trigger a transaction abort, regardless of whether it hits cache or DRAM.

To perform the attack, the attacker threads need a minimum cache eviction set. A minimum cache eviction set is a set of virtual addresses with which a user thread can make sure the target cache line is evicted out of the cache. For example, if the virtual address of the clock variable address on a CPU with 4-way associative LLC is `0x00007E30`, then the attacker could find an eviction set of addresses that shares the same cache set: {`0x00013D30`, `0x00026A30`, `0x000E2730`, `0x0009AB30`}, and according to the Intel cache structure, this means they should also reside on the same LLC cache slice. The allocation of LLC matters because cache entries on different cache slices do not belong to the same cache set. After finding the eviction set, to make sure that the clock variable is evicted from all levels of cache, one just needs to access all the addresses in the eviction set. Once the cache entry is successfully evicted, the secure timer thread needs to hit the DRAM to read or write to the clock variable, which slows down the increment speed of the software timer ticks.

We now explain how the cache eviction set can be found. As in Figure 2, the physical address of each memory request is decomposed into three parts when mapping the address to an LLC cache line. The part with the lowest bits of the address indicates the offset in the line, and the set bits decide which cache set it is mapped to. In recent Intel CPUs, LLC are further divided into slices, and an undocumented hash function maps the set and tag bits of the addresses to a specific LLC cache slice. While the hash function itself is undocumented, there have been attempts [14] to reverse-engineer it. Alternatively, other methods [22] can successfully find a minimum eviction set with user-level programs with high probability. We adopt existing methods for finding an LLC eviction set and use them for our attack.

After finding a cache eviction set, as shown in Figure 3, the attacker thread can then access all the virtual addresses in the eviction set of the clock variable in the software timer thread, so that the clock variable is evicted from the cache and the incrementing speed is much slower. The attacker can repeatedly access the eviction set and keep evicting the clock variable in a loop, so that whenever the software timer thread accesses the clock variable again and makes it cached, the cache entry will actively be evicted by the attacker again.

Because the attacker thread runs concurrently with the software timer thread, the eviction of the cache line containing the clock variable is probabilistic without the knowledge of the exact hardware cache replacement algorithm used by the CPU. However, the attacker can also improve the chance of cache eviction by parallelizing the accesses to the cache eviction set. The attackers can distribute the elements of the cache eviction set to different threads that are controlled by the attacker, preferably filling all the rest of available CPU cores with attacker threads. This approach turns the single-threaded attack into a multi-threaded coordinated attack and gives the attacker a better chance to evict the target victim cache entry more efficiently.

The attacker achieves the maximum timer slow-down effect by ideally forcing every increment of the clock variable to miss all levels of cache and hit DRAM. In this way, the software timer runs slower in comparison with the wall time at

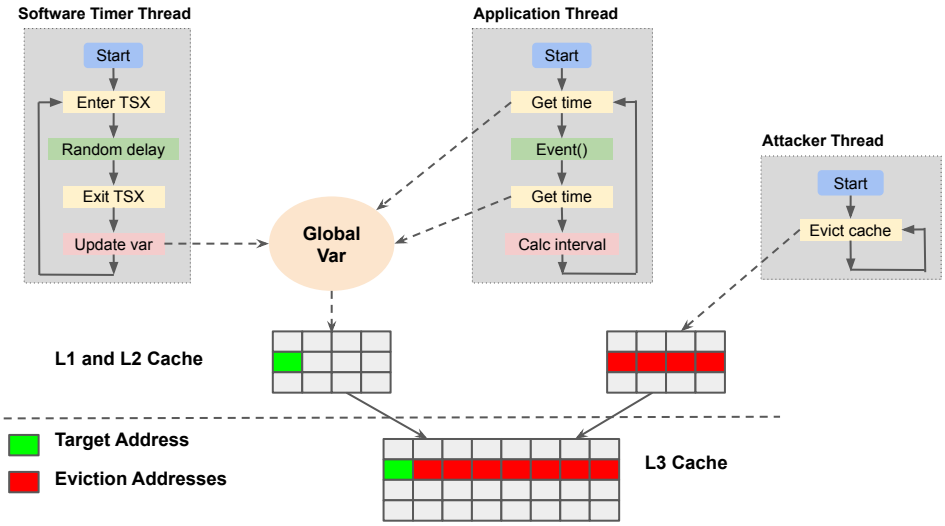

Fig. 3: *Illustration of Aion-2: Cache eviction attack.*

the maximum limit, which is the theoretical worst case for the reference timer. However, due to the nature of multi-core and scheduler, it is difficult for an attacker with only user-level privilege, to achieve this guarantee. We will show in the evaluation section how practical the attack is and describe our results.

To summarize, the target of both types of the Aion attacks is to manipulate the accuracy of the software timer, either to slow it down or speed it up without being noticed by the victim system. According to our software timer model, it is the accuracy measure M_t that the attacker focuses on. To attack M_t, the malicious party may either change the execution speed of CPU F_{CPU}, or the time $T_{Inc(V_G)}$ needed to increase the global clock variable.

5 Implementation

5.1 Reference Software Timer

In our experiments on Aion attacks, we use a real-world software timer as an implementation of the general model described in Section 3. We choose the software timer implementation from Déjà Vu [5], because (1) it has high accuracy for event rate measurement, and (2) it can detect repeated interruptions and protect itself from malicious preemptions from privileged threads. It not only includes an essential loop that increments the clock variable, but also additional defense code using Intel TSX that protects the software timer threads from frequent interruptions by malicious attackers as shown in Listing 1.1:
L1: The timer thread starts an infinite loop from an SGX enclave.
L2: It enters into a TSX-protected zone, where any interruption to the middle

```
1  while (infinite_loop_flag) {
     if (_xbegin() == _XBEGIN_STARTED) {   /* TSX begins */
3      __asm volatile {
         "rdrand %0\n\t"
5        :"=r"(rand)
       };
7      rand = (rand & 0x7) + 1;
       for (i = 0; i < rand; i++) {
9        for (k = 0; k < 5; k++)
           my_udelay(1);
11     }
       _xend();          /* TSX ends */
13   } else {
       int_flag++;
15   }
     current_time = current_time + rand;
17 }
```

Listing 1.1: Reference Timer Thread Implementation in C

of the TSX zone will fall into a trap, generate an exception, and rollback to the beginning of the entry point.
L3–7: It generates a random integer number between 1 and 8. Here the randomness is provided by `rdrand` as the original authors use it, while other pseudo-random functions could also work.
L8–11: This is a loop creating a delay proportional to the generated random value, so it is harder for attackers to guess when TSX protection covers the thread execution.
L12 and **L**16: The code leaves the TSX-protected region and the clock variable is updated. As mentioned in Section 4, the reason for ending the TSX zone before the timer tick number is updated is that the clock variable is intended to be read from other threads concurrently. If the update is in TSX zone then any concurrent read will abort the transaction and roll back the timer thread.

The clock variable (`current_time` in **L**16) is exposed to both the timer thread and the application threads. With some randomness, the timer thread periodically increment the clock variable. When the application thread needs to take a high-resolution time measurement of a particular event, it first retrieves the clock variable's value before the event and reads the same variable again afterward to calculate the interval. In this standard procedure, the TSX protection does not apply to the clock variable. Thus, theoretically, anyone can access it without triggering the TSX or SGX trap as long as they are in the enclave. However, the accuracy of the timer is questionable for two reasons. First, the thread execution speed is relevant to the CPU clock speed, because the real-world time of instructions being processed by the CPU and on which the clock speed is not known or controlled by the enclave applications. Secondly, the time of access of

the clock variable, whether from the timer thread or any other thread, cannot be assumed to be constant because there is no guarantee of which level of cache or memory it may hit. Therefore, the software timer is not as reliable and secure as was thought, even if it is running in an SGX enclave.

5.2 Implementing Aion-1: CPU Thermal and Frequency Attack

This type of attack focuses on changing the speed of targeted CPU cores. The two methods manipulating CPU core speed include triggering thermal events to force a core into HDC mode, and adjusting CPU frequency directly via the power control module of the OS. Because HDC mode stops a CPU core from running in a certain percentage of the time, and therefore both methods can be regarded equivalent to making a CPU core running at a certain frequency, we refer to it as the "equivalent frequency" later in this section.

The thermal management attack needs to be implemented in a thread with root privilege. We trigger thermal events by changing the respective MSRs: IA32_CLOCK_MODULATION and operate on the programmable bits of [3:1]. For direct frequency adjustment, there are three kernel modules we can use for scaling the CPU core frequencies: `intel_pstate` , `acpi-cpufreq` , and `p4-clockmod`. We have tested them all and found that 1) `intel_pstate` as a new power management module cannot achieve per-core frequency scaling, and 2) `p4-clockmod` as a relative last-generation kernel driver has dependency on the Intel `speedstep-lib` driver which is not compatible with our test CPUs. We therefore chose `acpi-cpufreq` as the driver that facilitates the attacker thread.

We set the kernel driver to use a "userspace" power governor, so that a user-level application with root privilege can configure any CPU core to run at a specified frequency. In this case, it is the attacker thread that controls the CPU core frequency of the software timer thread and other threads. Many methods can be used to trigger CPU thermal events, such as configuring TCC offset of CPU, increasing CPU workloads to stress out the cores, blocking physical airflow or stopping case fan from working, and sending software signal to CPU to force clock modulation. We choose the last approach that only requires a write to an MSR register `0x19A` for implementation, however, we believe that an attacker can use various creative approaches to generate thermal events.

In the attacker thread, we set the target execution speed for the software timer thread to F_c and other threads to F_x. The attacking thread first gets the information of which CPU core the software timer thread executes on, and then runs in a loop while `Loop_Flag` is `TRUE` to set the frequencies of the software timer thread and other threads. The attacker thread stops when `Loop_Flag` is changed to `FALSE`.

5.3 Implementing Aion-2: Cache Eviction Attack

The cache eviction attack needs the address of the timer variable used in the software timer thread to find the cache eviction set. We here assume that the image of the victim SGX application is openly available to all, which makes sense

because it is supposed to be loaded by the OS into an SGX enclave. The address can be determined by doing a binary analysis on the application image.

Once the address `Addr_t` is found after loading the SGX application, the attack can start to slow down the software timer: First, the attacker thread finds a cache eviction set for `Addr_t`. This can be done by an unprivileged user-level process using a group reduction algorithm [22], or like in our experiment for the ease of implementation, use the page map and get the physical address `Addr_p` of `Addr_t` to find the cache eviction set directly.

As is shown in Listing 2, with the physical address, the undocumented hash function is required to determine which LLC cache slice an address belongs to. We obtain the hash function by reverse engineering using the algorithms mentioned in previous work [14]. Then, with the cache eviction set `EV_t` in hand, the attacker thread can loop accessing the addresses in the eviction set to keep evicting the software timer thread's clock variable out of all levels of cache. This will then slowing down the timer because every time it increments the clock variable, it should hit DRAM instead of the cache.

We have optimized the attack by parallelizing the attacking loop: the addresses in the eviction set `EV_t` can be further divided and assigned to multiple threads. The attacker threads can keep accessing the addresses in the same eviction set and evicting the target clock variable from the cache faster, because the multithreaded attacker still shares the same LLC and it should take less time for all addresses in `EV_t` to be accessed to evict the target address `Addr_t`.

6 Evaluation

6.1 Purpose and Experiment Setup

We conduct all experiments on two machines with different CPUs: (1) Intel i7-6700K with 4 cores; and (2) Intel Xeon E3-1230 v6 with 4 cores. For the system software environment setup, we use Intel SGX v2.11 SDK on top of Linux with kernel v5.4, and all of the machines have hyper-threading enabled. The experiments in this section are conducted to demonstrate the following:

- Software timers in SGX enclaves are vulnerable to Aion attacks, which can manipulate the reported clock time from outside the enclave.
- Without compromising the software timer of the defender, a representative cache-based side-channel attack will be detected and prevented from exploiting the victim applications.
- With the help of Aion attacks, the same side-channel attacks can evade detection by the defender.

In the remaining parts of this section, we first demonstrate experiments and results that show that Aion attacks can successfully manipulate software timers in SGX enclaves. We then present an end-to-end attack to show that our attacks can facilitate an existing cache-based side-channel attack on SGX enclaves, evading the detection of a defender based on a software timer.

	Xeon E3-1230v6				i7-6700K			
Random Func (+CPU Thermal)	**RD**	**RD+TA**	**TF**	**TF+TA**	**RD**	**RD+TA**	**TF**	**TF+TA**
Baseline	**256.3**	7.0 (37×)	**337.4**	8.7 (39×)	**225.9**	5.8 (39×)	**302.5**	7.9 (38×)
Single-thread (Cache Eviction)	156.2 (1.6×)	5.2 (49×)	181.5 (1.9×)	6.1 (55×)	148.9 (1.5×)	4.7 (48×)	148.1 (2.0×)	4.1 (73×)
Multi-thread (Cache Eviction)	94.3 (2.7×)	2.3 (111×)	54.6 (6.2×)	1.8 (187×)	90.6 (2.5×)	1.9 (120×)	41.7 (7.3×)	1.5 (202×)

Table 1: Results of software timer readings affected by the Aion attacks.

6.2 Aion Attack Evaluation

Both types of Aion attacks have the same goal of manipulating the running speed of the software timer, and their effectiveness will be evaluated in this section. As we previously analyzed, Aion attacks can assist the side-channel attackers in evading the detection of existing defensive mechanisms that rely on the software timer to be accurate. We evaluate the extent to which our attacks can speed up or slow down the software timer, as this determines the probability of a successful side-channel attack.

We combine the two types of Aion attacks and demonstrate their effectiveness in manipulating the software timer as a unit test. We test the reduction rate (how much the attack can slow down the software timer) by the CPU thermal attack and cache eviction attack under different settings.

The baseline workload runs in an SGX enclave as a simple loop that runs operations from AES encryption. We compare the time intervals under different scenarios in Table 1, including: a) the baseline scenario where the timer is not under attack, and only affected by the thermal attack at the row of "baseline"; b) a single-thread attacker scenario where only one attacking thread of Aion-2 attack is running; and c) a multi-thread attackers scenario where the number of Aion-2 attacking threads is the `(# of total hyper threads - 2)`, and the other two threads are taken by the software timer thread and the application thread.

Another variant we are comparing in the evaluation shown in Table 1 is the different random functions used in the software timer thread. Because randomness is not free of cost, all random number generators take different amounts of execution time which may affect how much time the software timer thread spends in a loop to increment the clock variable. We use four different sets of functions for randomness generation and combined them with or without CPU core execution speed manipulation by the thermal attack: (1) `RD`: `RDRAND` instruction; (2) `RD+TA`, which combines CPU thermal attack and use it under (1)'s settings of `RDRAND` instruction; (3) `TF`, which is a simple pseudo-random number generator called T-Function [12]; and (4) `TF+TA`, which combines the CPU thermal attack and uses it under the same settings of (3) for the evaluation. We note here that since the `T-Function()` is so cheap in execution costing less than 30

cycles after optimization, that we did not repeat a separate evaluation with no randomness generated, which also makes sense in the scenario of real defence. Again, we make sure that the software timer thread, the application thread, and the attacking thread(2) in our evaluation do not share the same core to avoid them from competing for the same processor resource. After each number, the number in a bracket (e.g., 202×) indicates its slow-down factor comparing with the baseline.

From the above results, we can see that under various settings, the CPU thermal attack is powerful and can achieve 30-40× slow-down on its own. Also, both single-threaded and multi-threaded attacking methods can effectively slow down the software timer via the cache eviction attack, and in all but the RD case, achieve a slow down that exceeds the range of slowdowns that previous systems claim to be able to defend. Moreover, when combined with the core frequency manipulation attack, the effectiveness is further improved, in total slowing down the software timer by a factor greater than 200 for software timer using `T-function()` under both types of Aion attacks, and by a factor about 120 for software timer using `RDRAND` instruction under both types of Aion attacks.

6.3 End-to-End Attack Evaluation

We previously showed how much the Aion attacks can slow down the software timer, but the timer slow-down ratio alone may not be enough evidence to prove that the software timer slow-down rate can effectively assist other side-channel attacks to go undetected by SGX side-channel defenders. Therefore, for a full evaluation, we have mounted an end-to-end attack to demonstrate the complete procedure, combining the traditional SGX side-channel attack and our Aion attacks to defeat the software timer that defenders rely on. The end-to-end attack experiments consist of three parts: (1) a known side-channel attack on SGX; (2) a defender used to detect the side-channel attack in (1); and (3) our Aion attacks that can compromise the defensive mechanism in (2).

In our experiments, the side-channel attack is implemented based on the SGX-Step framework [21] and uses a Prime+Probe [13] type cache-based side-channel attack to extract an AES key that is used to do encryption operations repeatedly inside an SGX enclave. We use the OpenSSL 0.9.7a library, which is known to be vulnerable against cache timing attacks and is also known to work in the environment of SGX, as a proof-of-concept demonstration. The ratio of successful key extractions of the attack on our Intel i-7 6700K machine is above 98.4% under 100K rounds of victim encryption operations.

For the defender, we have tested our implementation of SGX side-channel attack defender based on the defense paper [5]. Our results are comparable to the evaluation data shown in the original work: the defender can successfully detect at least 95% of the basic SGX side-channel attacks under a trained threshold value δ. The threshold value is gathered and calculated by running normal applications in SGX enclaves without being attacked, so it counts and tracks the normal number of AEX events happening and decides to trigger the alarm when there is a burst of an abnormal amount of such events. The results with the Intel

Benchmark	Baseline Defence			Defence Under Aion Attack		
	Threshold	Acc %	FP%	Threshold	Acc % (E3)	Acc % (i7)
Numeric sort	4	100	97	4	95	94
	40	100	40	40	17	15
	80	**95**	**3**	**80**	**2**	**2**
	160	87	2	160	1	0
	320	40	0	320	0	0
	640	9	0	-	-	-
	1280	3	0	-	-	-
Fourier	4	100	98	4	95	92
	40	100	46	40	19	18
	80	**96**	**4**	**80**	**2**	**1**
	160	74	2	160	0	0
	320	30	0	320	0	0
	640	10	0	-	-	-
	1280	2	0	-	-	-

Table 2: End-to-end evaluation of Aion attacks against existing defences.

i7-6700K machine environment are shown in the column of Baseline Defence in Table 2 with the randomness generator of `RDRAND` in its software timer loop.

Distinct from the previous evaluation that shows the effect of Aion attacks on the software timer, the evaluation with end-to-end attacks combines our two types of Aion attacks with the basic side-channel attacks, and the SGX side-channel attack defender. We measure how effectively our attacks can assist the base side-channel attack to evade detection of the defender. Experiments run on two of the machines with SGX with the defender using `RDRAND` as the randomness generator, using both types of Aion attacks combined, and results are shown in Table 2. From the results, we can see that under Aion attacks, the defender identifies less than 2% of the side-channel attacks in its normal setting of the threshold value 80. For the threshold value of 40, the accuracy is less than 15–19%, however, without the knowledge of the ongoing Aion attacks, the defender would not choose to use a low threshold value by taking the risk of a high false-positive rate. The results demonstrate that the combined Aion attacks can effectively assist the base side-channel SGX attack to make it undetectable by the SGX defensive software that is based on a software timer.

To summarize our findings, software timer-based defenses are not viable in the face of tampering of timers from Aion attacks. Comparing the left part of Table 2, which shows false positive rates under benign conditions at various E_{th} thresholds, with the right part of Table 2, which shows detection accuracy after tampering with the Aion attack, we can see that the detection rate is on the order of the false positive rate. At threshold 80, both of the false positive rates in benign conditions are 3-4% while the detection rates under attack are 1-2%. Even if we decrease the threshold to 40, the detection rates only range from 14-19% while the false positive rates have increased to 40-46%. As a result, it is not possible for the defender to select a threshold that permits good detection

when under attack, but still keeps false positives at acceptable levels. We see this trend holds for all thresholds. As a result, our empirical analysis shows that it is not possible to use a software timer in any defense due the adversary's ability to manipulate the timer.

7 Conclusion

Side-channel attacks are major threats that TEEs currently face, including Intel SGX. Although software-based defences have been proposed for detection and prevention of cache-based side-channel attacks, the lack of a reliable hardware timer for secure applications to use inside the enclave makes such solutions vulnerable against Aion attacks. In this paper, we design and implement two types of Aion attacks, one based on manipulating the software timer thread execution speed by triggering CPU thermal events, and the other focusing on cache eviction to slow down the rate at which the target timer is increased, both of which can effectively change the how fast the software timer in an SGX enclave runs and invalidate the defensive approaches that rely on accurate high-resolution software timers. We also argue that in our general software timer model, there is no way to design a reliable timer purely in software that makes the defense usable and effective in detecting side-channel attacks, unless the defence can tolerate up to 200x slowdown of their timer, but this is unlikely.

The core of the problem we show through our model analysis is that when system designers use a software timer to measure the time certain critical events take, they made an invalid assumption: That the increment speed of the variable used in the software timer is nearly constant, and can not be significantly altered by an adversary. However, this fails to account for dramatic changes in execution speed when accessing a global variable that can occur when the CPU frequency varies or when cache behaviour is manipulated. By breaking the high-resolution measurement of time, Aion attacks are able to exploit existing defences.

Acknowledgements. We would like to thank Professor Yinqian Zhang, Dr. Sanchuan Chen and Oleksii Oleksenko for their help in SGX defensive frameworks. We appreciate Dr. Lianying Zhao, Tony Liao and colleagues from Baidu Research for their valuable advice on this paper.

References

1. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software grand exposure: SGX cache attacks are practical. In: Proc. of USENIX WOOT. Vancouver, Canada (2017)
2. Brotzman, R., Liu, S., Zhang, D., Tan, G., Kandemir, M.: CaSym: Cache aware symbolic execution for side channel detection and mitigation. In: Proc. of IEEE S&P. San Fransico, USA (2019)
3. Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: Proc. of USENIX Security. Baltimore, USA (August 2018)
4. Bulck, J.V., Weichbrodt, N., Kapitza, R., Piessens, F., Strackx, R.: Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In: Proc. of USENIX Security. Vancouver, Canada (2017)

5. Chen, S., Zhang, X., Reiter, M.K., Zhang, Y.: Detecting privileged side-channel attacks in shielded execution with Déjà Vu. In: Proc. of ASIA CCS. Abu Dhabi, UAE (2017)
6. Costan, V., Devadas, S.: Intel SGX explained. Cryptology ePrint Archive, Report 2016/086 (2016), URL: https://ia.cr/2016/086
7. Dall, F., Micheli, G., Eisenbarth, T., Genkin, D., Heninger, N., Moghimi, A., Yarom, Y.: CacheQuote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. In: Proc. of CHES. Amsterdam, Netherlands (2018)
8. Gözfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache attacks on Intel SGX. In: Proc. of EuroSec. Belgrade, Serbia (2017)
9. Guan, L., Lin, J., Luo, B., Jing, J., Wang, J.: Protecting private keys against memory disclosure attacks using hardware transactional memory. In: Proc. of IEEE S&P. San Jose, USA (2015)
10. Hähnel, M., Cui, W., Peinado, M.: High-Resolution side channels for untrusted operating systems. In: Proc. of USENIX ATC. Santa Clara, USA (2017)
11. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual.
12. Klimov, A., Shamir, A.: A new class of invertible mappings. In: Proc. of CHES. San Francisco Bay, CA, USA (2002)
13. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: Proc. of IEEE S&P. San Jose, USA (2015)
14. Maurice, C., Scouarnec, N., Neumann, C., Heen, O., Francillon, A.: Reverse engineering intel last-level cache complex addressing using performance counters. In: Proc. of RAID. Kyoto, Japan (2015)
15. Moghimi, A., Irazoqui, G., Eisenbarth, T.: CacheZoom: How SGX amplifies the power of cache attacks. In: Proc. of CHES. Taipei, Taiwan (2017)
16. Neve, M., Seifert, J.P.: Advances on access-driven cache attacks on AES. In: Proc. of SAC workshop. pp. 147–162. Montreal,Canada (2006)
17. Oleksenko, O., Trach, B., Krahn, R., Martin, A., Fetzer, C., Silberstein, M.: Varys: Protecting SGX enclaves from practical side-channel attacks. In: Proc. of USENIX ATC. Boston, USA (2018)
18. Qureshi, M.K.: CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In: Proc. of IEEE MICRO. Fukuoka, Japan (2018)
19. Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware Guard Extension: Using SGX to conceal cache attacks. In: Proc. of DIMVA. Bonn, Germany (2017)
20. Shih, M.W., Lee, S., Kim, T., Peinado, M.: T-SGX: Eradicating controlled-channel attacks against enclave programs. In: Proc. of NDSS. San Diego, USA (2017)
21. Van Bulck, J., Piessens, F., Strackx, R.: SGX-Step: A practical attack framework for precise enclave execution control. In: Proc. of SysTEX. Shanghai, China (2017)
22. Vila, P., Kopf, B., Morales, J.F.: Theory and practice of finding eviction sets. In: Proc. of IEEE S&P. pp. 39–54. San Francisco, USA (2019)
23. Wang, S., Wang, P., Liu, X., Zhang, D., Wu, D.: CacheD: Identifying cache-based timing channels in production software. In: Proc. of USENIX Security. Vancouver, Canada (2017)
24. Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., Gunter, C.A.: Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In: Proc. of CCS. Dallas, USA (2017)
25. Xu, Y., Cui, W., Peinado, M.: Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: Proc. of IEEE S&P. San Jose, USA (2015)
26. Yarom, Y., Falkner, K.: Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In: Proc. of USENIX Security. San Diego, USA (2014)