Attacking Inclusive Last-Level Caches Automatically

Ms. PRIYADARSHNI SAMAL, RAJESH KUMAR PATI*
Dept. OF Computer Science and Engineering, NIT , BBSR
priyadarsini@thenalanda.com,rajeshkumar@thenalanda.com*

Abstract

Recent research on cache attacks has demonstrated that CPU caches are a significant information leakage source. However, current attacks necessitate manually identifying weaknesses, such as data accesses or instruction executions dependent on confidential information. We discuss cache template attacks in this article. With this general attack method, we can automatically profile and take advantage of any program's cache-based information leakage without having any prior knowledge of the software or even the system. Without requiring any prior offline calculations or measurements, Cache Template Attacks can be carried out online on a remote machine. Attacks using cache templates have two stages. We identify connections between processing secret data, such as particular key inputs or private keys of cryptographic primitives, and particular cache accesses during the profiling phase. We calculate the secret values at the exploitation stage using observed cache accesses. We demonstrate the effectiveness of the suggested strategy in a number of attacks as well as in a practical application for programmers. The application of Cache Template Attacks to infer keystrokes and—even more serious—the identification of certain keys on Linux and Windows user interfaces are among the techniques that are demonstrated. More particular, we can drop the entropy per character on Linux platforms for lower-case only passwords from log2(26)= 4.7 to 1.4 bits. Also, we launch an automated attack.

Introduction

Cache-based side-channel attacks have gained increasing attention among the scientific community. First, in terms of ever improving attacks against cryptographic implementations, both symmetric [4, 6, 16, 39, 41, 53] as well as asymmetric cryptography [3, 7, 9, 54], and sec-, in terms of developing countermeasures to prevent these types of attacks [31, 34]. Recently, Yarom and Falkner [55] proposed the Flush+Reload attack, which has been successfully applied against cryptographic implementations [3, 17, 22]. Besides the possibility of attacking cryptographic implementations, Yarom and Falkner pointed out that their attack might also be used to attack other software as well, for instance, to collect keystroke timing information. However, no clear indication is given on how to exploit such vulnerabilities with their attack. A similar attack has already been suggested in 2009 by Ristenpart et al. [44], who reported being able to gather keystroke timing information by observing cache activities on an otherwise idle machine.

The limiting factor of all existing attacks is that sophisticated knowledge about the attacked algorithm or software is necessary, i.e., access to the source code or even modification of the source code [7] is required in order to identify vulnerable memory accesses or the execution of specific code fragments manually.

In this paper, we make use of the Flush+Reload attack [55] and present the concept of *Cache Template Attacks*,[1] a generic approach to exploit cache-based vulnerabilities in any program running on architectures with shared inclusive last-level caches. Our attack exploits four fundamental concepts of modern cache architectures and operating systems.

1. Last-level caches are shared among all CPUs.
2. Last-level caches are inclusive, i.e., all data which is cached within the L1 and L2 cache must also be cached in the L3 cache. Thus, any modification of the L3 cache on one core immediately influences the cache behavior of all other cores.
3. Cache lines are shared among different processes.
4. The operating system allows programs to map any other program binary or library, i.e., code and static data, into their own address space.

[1]The basic framework can be found at https://github.com/IAIK/cache_template_attacks.

Based on these observations, we demonstrate how to perform Cache Template Attacks on any program automatically in order to determine memory addresses which are accessed depending on secret information or specific events. Thus, we are not only able to attack cryptographic implementations, but also any other event, e.g., keyboard input, which might be of interest to an attacker.

We demonstrate how to use Cache Template Attacks to derive keystroke information with a deviation of less than 1 microsecond from the actual keystroke and an accuracy of almost 100%. With our approach, we are not only able to infer keystroke timing information, but even to infer specific keys pressed on the keyboard, both for GTK-based Linux user interfaces and Windows user interfaces. Furthermore, all attacks to date require sophisticated knowledge of the attacked software and the executable itself. In contrast, our technique can be applied to any executable in a generic way. In order to demonstrate this, we automatically attack the T-table-based AES [10, 35] implementation of OpenSSL [37].

Besides demonstrating the power of Cache Template Attacks to exploit cache-based vulnerabilities, we also discuss how this generic concept supports developers in detecting cache-based information leaks within their own software, including third party libraries. Based on the insights we gained during the development of the presented concept, we also present possible countermeasures to mitigate specific types of cache attacks.

Outline. The remaining paper is organized as follows. In Section 2, we provide background information on CPU caches, shared memory, and cache attacks in general. We describe Cache Template Attacks in Section 3. We illustrate the basic idea on an artificial example program in Section 4 and demonstrate Cache Template Attacks against real-world applications in Section 5. In Section 6, we discuss countermeasures against cache attacks in general. Finally, we conclude in Section 7.

1 Background and Related Work

In this section, we give a basic introduction to the concept of CPU caches and shared memory. Furthermore, we provide a basic introduction to cache attacks.

CPU Caches

The basic idea of CPU caches is to hide memory accesses to the slow physical memory by buffering frequently used data in a small and fast memory. Today, most architectures employ set-associative caches, meaning that the cache is divided into multiple cache sets and each cache set consists of several cache lines (also called ways). An index is used to map specific memory locations to the sets of the cache memory.

We distinguish between virtually indexed and physically indexed caches, which derive the index from the virtual or physical address, respectively. In general, virtually indexed caches are considered to be faster than physically indexed caches. However, the drawback of virtually indexed caches is that different virtual addresses mapping to the same physical address are cached in different cache lines. In order to uniquely identify a specific cache line within a cache set, so-called tags are used. Again, caches can be virtually tagged or physically tagged. A virtual tag has the same drawback as a virtual index. Physical tags, however, are less expensive than physical indices as they can be computed simultaneously with the virtual index.

In addition, there is a distinction between inclusive and exclusive caches. On Intel systems, the L3 cache is an inclusive cache, meaning that all data within the L1 and L2 caches are also present within the L3 cache. Furthermore, the L3 cache is shared among all cores. Due to the shared L3 cache, executing code or accessing data on one core has immediate consequences for all other cores. This is the basis for the Flush+Reload [55] attack as described in Section 2.3.

Our test systems (Intel Core i5-2/3 CPUs) have two 32 KB L1 caches—one for data and one for instructions—per core, a unified L2 cache of 256 KB, and a unified L3 cache of 3 MB (12 ways) shared among all cores. The cache-line size is 64 bytes for all caches.

Shared Memory

Operating systems use shared memory to reduce memory utilization. For instance, libraries used by several programs are shared among all processes using them. The operating system loads the libraries into physical memory only once and maps the same physical pages into the address space of each process.

The operating system employs shared memory in several more cases. First, when forking a process, the memory is shared between the two processes. Only when the data is modified, the corresponding memory regions are copied. Second, a similar mechanism is used when starting another instance of an already running program. Third, it is also possible for user programs to request shared memory using system calls like mmap.

The operating system tries to unify these three categories. On Linux, mapping a program file or a shared library file as a read-only memory with mmap results in sharing memory with all these programs, respectively programs using the same shared library or program binary. This is also possible on Windows using the LoadLibrary function. Thus, even if a program is stat-

ically linked, its memory is shared with other programs which execute or map the same binary.

Another form of shared memory is content-based page deduplication. The hypervisor or operating system scans the physical memory for pages with identical content. All mappings to identical pages are redirected to one of the pages while the other pages are marked as free. Thus, memory is shared between completely unrelated processes and even between processes running in different virtual machines. When the data is modified by one process, memory is duplicated again. These examples demonstrate that code as well as static data can be shared among processes, even without their knowledge. Nevertheless, page deduplication can enhance system performance and besides the application in cloud systems, it is also relevant in smaller systems like smartphones.

User programs can retrieve information on their virtual and physical memory using operating-system services like /proc/<pid>/maps on Linux or tools like vmmap on Windows. The list of mappings typically includes all loaded shared-object files and the program binary.

Cache Attacks

Cache attacks are a specific type of side-channel attacks that exploit the effects of the cache memory on the execution time of algorithms. The first theoretical attacks were mentioned by Kocher [28] and Kelsey et al. [26]. Later on, practical attacks for DES were proposed by Page [41] as well as Tsunoo et al. [50]. In 2004, Bernstein [4] proposed the first time-driven cache attack against AES. This attack has been investigated quite extensively [36].

A more fine-grained attack has been proposed by Percival [42], who suggested to measure the time to access all ways of a cache set. As the access time correlates with the number of occupied cache ways, an attacker can determine the cache ways occupied by other processes. At the same time, Osvik et al. [39] proposed two fundamental techniques that allow an attacker to determine which specific cache sets have been accessed by a victim program. The first technique is Evict+Time, which consists of three steps. First, the victim program is executed and its execution time is measured. Afterwards, an attacker evicts one specific cache set and finally measures the execution time of the victim again. If the execution time increased, the cache set was probably accessed during the execution.

The second technique is Prime+Probe, which is similar to Percival's attack. During the Prime step, the attacker occupies specific cache sets. After the victim program has been scheduled, the Probe step is used to determine which cache sets are still occupied.

Later on, Gullasch et al. [16] proposed a significantly more powerful attack that exploits the fact that shared memory is loaded into the same cache sets for different processes. While Gullasch et al. attacked the L1 cache, Yarom and Falkner [55] presented an improvement called Flush+Reload that targets the L3 cache.

Flush+Reload relies on the availability of shared memory and especially shared libraries between the attacker and the victim program. An attacker constantly flushes a cache line using the clflush instruction on an address within the shared memory. After the victim has been scheduled, the attacker measures the time it takes to reaccess the same address again. The measured time reveals whether the data has been loaded into the cache by reaccessing it or whether the victim program loaded the data into the cache before reaccessing. This allows the attacker to determine the memory accesses of the victim process. As the L3 cache is shared among all cores, it is not necessary to constantly interrupt the victim process. Instead, both processes run on different cores while still working on the same L3 cache. Furthermore, the L3 cache is a unified inclusive cache and, thus, even allows to determine when a certain instruction is executed. Because of the size of the L3 cache, there are significantly fewer false negative cache-hit detections caused by evictions. Even though false positive cache-hit detections (as in Prime+Probe) are not possible because of the shared-memory-based approach, false positive cache hits can still occur if data is loaded into the cache accidentally (e.g., by the prefetcher). Nevertheless, applications of Flush+Reload have been shown to be quite reliable and powerful, for example, to detect specific versions of cryptographic libraries [23], to revive supposedly fixed attacks (e.g., Lucky 13) [24] as well as to improve attacks against T-table-based AES implementations [17].

As shared memory is not always available between different virtual machines in the cloud, more recent cache attacks use the Prime+Probe technique to perform cache attacks across virtual machine borders. For example, Irazoqui et al. [20] demonstrated a cross-VM attack on a T-Table-based AES implementation and Liu et al. [32] demonstrated a cross-VM attack on GnuPG. Both attacks require manual identification of exploitable code and data in targeted binaries. Similarly, Maurice et al. [33] built a cache-index-agnostic cross-VM covert channel based on Prime+Probe.

Simultaneous to our work, Oren et al. [38] developed a cache attack from within sandboxed JavaScript to attack user-specific data like network traffic or mouse movements. Contrary to existing attack approaches, we present a general attack framework to exploit cache vulnerabilities automatically. We demonstrate the effectiveness of this approach by inferring keystroke information and, for comparison reasons, by attacking a T-table-based AES implementation.

2 Cache Template Attacks

Chari et al. [8] presented template attacks as one of the strongest forms of side-channel attacks. First, side-channel traces are generated on a device controlled by the attacker. Based on these traces, the template—an exact model of signal and noise—is generated. A single side-channel trace from an identical device with unknown key is then iteratively classified using the template to derive the unknown key.

Similarly, Brumley and Hakala [7] described cache-timing template attacks to automatically analyze and exploit cache vulnerabilities. Their attack is based on Prime+Probe on the L1 cache and, thus, needs to run on the same core as the spy program. Furthermore, they describe a profiling phase for specific operations executed in the attacked binary, which requires manual work or even modification of the attacked software. In contrast, our attack only requires an attacker to know how to trigger specific events in order to attack them. Subsequently, Brumley and Hakala match these timing templates against the cache timing observed. In contrast, we match memory-access templates against the observed memory accesses.

Inspired by their work we propose *Cache Template Attacks*. The presented approach of Cache Template Attacks allows the exploitation of any cache vulnerability present in any program on any operating system executed on architectures with shared inclusive last-level caches and shared memory enabled. Cache Template Attacks consist of two phases: 1) a profiling phase, and 2) an exploitation phase. In the profiling phase, we compute a Cache Template matrix containing the cache-hit ratio on an address given a specific target event in the binary under attack. The exploitation phase uses this Cache Template matrix to infer events from cache hits.

Both phases rely on Flush+Reload and, thus, attack code and static data within binaries. In both phases the attacked binary is mapped into read-only shared memory in the attacker process. By accessing its own virtual addresses in the allocated read-only shared memory region, the attacker accesses the same physical memory and the same cache lines (due to the physically-indexed last level cache) as the process under attack. Therefore, the attacker completely bypasses address space layout randomization (ASLR). Also, due to shared memory, the additional memory consumption caused by the attacker process is negligible, i.e., in the range of a few megabytes at most.

In general, both phases are performed online on the attacked system and, therefore, cannot be prevented through differences in binaries due to different versions or the concept of software diversity [12]. However, if online profiling is not possible, e.g., in case the events must be triggered by a user or Flush+Reload is not possible on the attacked system, it can also be performed in a controlled environment. Below, we describe the profiling phase and the exploitation phase in more detail.

Profiling Phase

The profiling phase measures how many cache hits occur on a specific address during the execution of a specific event, i.e., the *cache-hit ratio*. The cache-hit ratios for different events are stored in the Cache Template matrix which has one column per event and one row per address. We refer to the column vector for an event as a *profile*. Examples of Cache Template matrices can be found in Section 4 and Section 5.1.

An *event* in terms of a Cache Template Attack can be anything that involves code execution or data accesses, e.g., low-frequency events, such as keystrokes or receiving an email, or high-frequency events, such as encryption with one or more key bits set to a specific value. To automate the profiling phase, it must be possible to trigger the event programmatically, e.g., by calling a function to simulate a keypress event, or executing a program.

The Cache Template matrix is computed in three steps. The first step is the generation of the cache-hit trace and the event trace. This is the main computation step of the Cache Template Attack, where the data for the Template is measured. In the second step, we extract the cache-hit ratio for each trace and store it in the Cache Template matrix. In a third post-processing step, we prune rows and columns which contain redundant information from the matrix. Algorithm 1 summarizes the profiling phase. We explain the corresponding steps in detail below.

Algorithm 1: Profiling phase.

Input: Set of events E, target program binary B, duration d
Output: Cache Template matrix T

```
Map binary B into memory
foreach event e in E do
    foreach address a in binary B do
        while duration d not passed do
            simultaneously
            Trigger event e and save event trace g^(E)_{a,e}
            Flush+Reload attack on address a
                and save cache-hit trace g^(H)_{a,e}
        end
        Extract cache-hit ratio H_{a,e} from g^(E)_{a,e}
            and g^(H)_{a,e} and store it in T
    end
end
Prune Cache Template matrix T
```

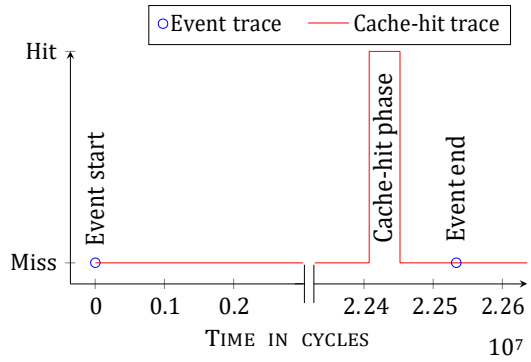

Figure 1: Trace of a single keypress event for address 0x4ebc0 of libgdk.so.

Cache-Hit Trace and Event Trace. The generation of the cache-hit trace and the event trace is repeated for each event and address for the specified duration (the while loop of Algorithm 1). The cache-hit trace $g^{(H)}_{a,e}$ is a binary function which has value 1 for every timestamp t where a cache hit has been observed. The function value remains 1 until the next timestamp t where a cache miss has been observed. We call subsequent cache hits a cache-hit phase. The event trace $g^{(E)}_{a,e}$ is a binary function which has value 1 when the processing of one specific event e starts or ends and value 0 for all other points.

In the measurement step, the binary under attack is executed and the event is triggered constantly. Each address of the attacked binary is profiled for a specific duration d. It must be long enough to trigger one or more events. Therefore, d depends only on the execution time of the event to be measured. The more events triggered within the specified duration d, the more accurate the resulting profile is. However, increasing the duration d increases the overall time required for the profiling phase.

The results of this measurement step are a cache-hit trace and an event trace, which are generated for all addresses a in the binary and all events e we want to profile. An excerpt of such a cache-hit trace and the corresponding event trace is shown in Figure 1. The start of the event is measured directly before the event is triggered. As we monitor library code, the cache-hit phase is measured before the attacked binary observes the event.

The generation of the traces can be sped up by two factors. First, in case of a cache miss, the CPU always fetches a whole cache line. Thus, we cannot distinguish between offsets of different accesses within a cache line and we can deduce the same information by probing only one address within each cache-line sized memory area.

Second, we reduce the overall number of triggered events by profiling multiple addresses at the same time. However, profiling multiple addresses on the same page can cause prefetching of more data from this page. Therefore, we can only profile addresses on different pages simultaneously. Thus, profiling all pages only takes as long as profiling a single page.

In case of low-frequency events, it is possible to profile all pages within one binary in parallel. However, this may lead to less accurate cache-hit traces $g^{(H)}_{a,e}$, i.e., timing deviations above 1 microsecond from the real event, which is only acceptable for low-frequency events.

Hit-Ratio Extraction. After the cache-hit trace and the event trace have been computed for a specific event e and a specific address a (the while loop of Algorithm 1), we derive the cache-hit ratio for each event and address. The cache-hit ratio $H_{a,e}$ is either a simple value or a time-dependent ratio function. In our case it is the ratio of cache hits on address a and the number of times the event e has been triggered within the profiling duration d.

To illustrate the difference between a cache-hit ratio with time dependency and without time dependency, we discuss two such functions. The cache-hit ratio with time dependency can be defined as follows. The event traces contain the start and end points of the processing of one event e. These start and end points define the relevant parts (denoted as *slices*) within the cache-hit trace. The slices are stored in a vector and scaled to the same length. Each slice contains a cache-hit pattern relative to the event e. If we average over this vector, we get the cache-hit ratio function for event e.

The second, much simpler approach is to define the cache-hit ratio without time dependency. In this case, we count the number of cache hits k on address a and divide it by the number of times n the event e has been triggered within the profiling duration d. That is, we define $H_a = \frac{k}{n}$. In case of a low-noise side channel and event detection through single cache hits, it is sufficient to use a simple hit-ratio extraction function.

Like the previous step, this step is repeated for all addresses a in the binary b and all events e to be profiled. The result is the full Cache Template matrix T. We denote the column vectors $\vec{p}_e$ as *profiles* for specific events.

Pruning. In the exploitation phase, we are limited regarding the number of addresses we can attack. Therefore, we want to reduce the number of addresses in the Cache Template. We remove redundant rows from the Cache Template matrix and merge events which cannot be distinguished based on their profiles $\vec{p}_e$.

As cache hits can be independent of an event, the measured cache-hit ratio on a specific address can be independent of the event, i.e., code which is always executed, frequent data accesses by threads running all the time, or code that is never executed and data that is never accessed. In order to be able to detect an event e, the set

of events has to contain at least one event e' which does not include event e. For example, in order to be able to detect the event "user pressed key A" we need to profile at least one event where the user does not press key A.

The pruning happens in three steps on the matrix. First, the removal of all addresses that have a small difference between minimum and maximum cache-hit ratio for all events. Second, merging all similar columns (events) into one set of events, i.e., events that cannot be distinguished from each other are merged into one column. The similarity measure for this is, for example, based on a mean squared error (MSE) function. Third, the removal of redundant lines. These steps ensure that we select the most interesting addresses and also allows us to reduce the attack complexity by reducing the overall number of monitored addresses.

We measure the reliability of a cache-based side channel by true and false positives as well as true and false negatives. Cache hits that coincide with an event are counted as true positive and cache hits that do not coincide with an event as false positive. Cache misses which coincide with an event are counted as true negative and cache misses which do not coincide with an event as false negative. Based on these four values we can determine the accuracy of our Template, for instance, by computing the F-Score, which is defined as the harmonic mean of the cache-hit ratio and the positive predictive value (percentage of true positives of the total cache hits). High F-Score values show that we can distinguish the given event accurately by attacking a specific address. In some cases further lines can be pruned from the Cache Template matrix based on these measures. The true positive rate and the false positive rate for an event e can be determined by the profile $\vec{p}_e$ of e and the average over all profiles except e.

Runtime of the Profiling Phase. Measuring the cache-hit ratio is the most expensive step in our attack. To quantify the cost we give two examples. In both cases we want to profile a 1 MB library, once for a low-frequency event, e.g., a keypress, and once for a high-frequency event, e.g., an encryption. In both cases, we try to achieve a runtime which is realistic for offline and online attacks while maintaining a high accuracy.

We choose a profiling duration of $d = 0.8$ seconds for the low-frequency event. During 0.8 seconds we can trigger around 200 events, which is enough to create a highly accurate profile. Profiling each address in the library for 0.8 seconds would take 10 days. Profiling only cache-line-aligned addresses still takes 4 hours. Applying both optimizations, the full library is profiled in 17 seconds.

In case of the high-frequency event, we attack an encryption. We assume that one encryption and the corresponding Flush+Reload measurement take 520 cycles on average. As in the previous example, we profile each address 200 times and, thus, we need 40–50 microseconds per address, i.e., $d = 50\mu s$. The basic attack takes less than 55 seconds to profile the full library for one event. Profiling only cache-line-aligned addresses takes less than 1 second and applying both optimizations results in a negligible runtime.

As already mentioned above, the accuracy of the resulting profile depends on how many times an event can be triggered during profiling duration d. In both cases we chose durations which are more than sufficient to create accurate profiles and still achieve reasonable execution times for an online attack. Our observations showed that it is necessary to profile each event at least 10 times to get meaningful results. However, profiling an event more than a few hundred times does not increase the accuracy of the profile anymore.

Exploitation Phase

In the exploitation phase we execute a generic spy program which performs either the Flush+Reload or the Prime+Probe algorithm. For all addresses in the Cache Template matrix resulting from the profiling phase, the cache activity is constantly monitored.

We monitor all addresses and record whether a cache hit occurred. This information is stored in a boolean vector $\vec{h}$. To determine which event occurred based on this observation, we compute the similarity $S(\vec{h}, \vec{p}_e)$ between $\vec{h}$ and each profile $\vec{p}_e$ from the Cache Template matrix. The similarity measure S can be based, for example, on a mean squared error (MSE) function. Algorithm 2 summarizes the exploitation phase.

Algorithm 2: Exploitation phase.

Input: Target program binary b,
Cache Template matrix $T = (\vec{p}_{e_1}, \vec{p}_{e_2}, \ldots, \vec{p}_{e_n})$
Map binary b into memory
repeat
  foreach *address a in T* **do**
    Flush+Reload attack on address a
    Store 0/1 in $\vec{h}[a]$ for cache miss/cache hit
  end
  if $\vec{p}_e$ *equals* $\vec{h}$ *w.r.t. similarity measure* **then**
    Event e detected
  end

The exploitation phase has the same requirements as the underlying attack techniques. The attacker needs to be able to execute a spy program on the attacked system. In case of Flush+Reload, the spy program needs no privileges, except opening the attacked program binary in a read-only shared memory. It is even possible

```
1 int map[130][1024] = {{-1U},...,{-130U}};
2 int main(int argc, char** argv) {
3   while(1) {
4     int c = getchar(); // unbuffered
5     if (map[(c % 128) + 1][0] == 0)
6       exit(-1);
7 } }
```

Listing 1: Victim program with large array on Linux

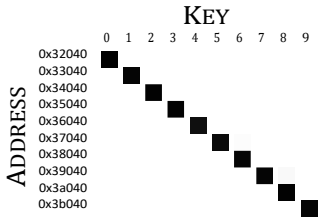

Figure 2: Cache Template matrix for the artificial victim program shown in Listing 1. Dark cells indicate high cache-hit ratios.

to attack binaries running in a different virtual machine on the same physical machine, if the hypervisor has page deduplication enabled. In case of Prime+Probe, the spy program needs no privileges at all and it is even possible to attack binaries running in a different virtual machine on the same physical machine, as shown by Irazoqui et al. [20]. However, the Prime+Probe technique is more susceptible to noise and therefore the exploitation phase will produce less reliable results, making attacks on low-frequency events more difficult.

The result of the exploitation phase is a log file containing all detected events and their corresponding timestamps. The interpretation of the log file still has to be done manually by the attacker.

4 Attacks on Artificial Applications

Before we actually exploit cache-based vulnerabilities in real applications in Section 5, we demonstrate the basic working principle of Cache Template Attacks on two artificial victim programs. These illustrative attacks show how Cache Template Attacks automatically profile and exploit cache activity in any program. The two attack scenarios we demonstrate are: 1) an attack on lookup tables, and 2) an attack on executed instructions. Hence, our ideal victim program or library either contains a large lookup table which is accessed depending on secret information, e.g., depending on secret lookup indices, or specific portions of program code which are executed based on secret information.

Attack on Data Accesses. For demonstration purposes, we spy on simple events like keypresses. In our victim program, shown in Listing 1, each keypress causes a memory access in a large array called `map`. These key-based accesses are 4096 bytes apart from each other to avoid triggering the prefetcher. The array is initialized with static values in order to place it in the data segment and to guarantee that each page contains different data and, thus, is not deduplicated in any way. It is necessary to place it in the data segment in order to make it shareable with the spy program.

In the profiling phase of the Cache Template Attack, we simulate different keystroke events using the X11 automation library `libxdo`. This library can be linked statically into the spy program, i.e., it does not need to be installed. The Cache Template matrix is generated as described in Section 3. Within a duration of $d = 0.8$ seconds we simulated around 700 keypress events. The resulting Cache Template matrix can be seen in Figure 2 for all number keys. We observe cache hits on addresses that are exactly 4 096 bytes apart, which is due to the data type and the dimension of the `map` array. In our measurements, there were less than 0.3% false positive cache hits on the corresponding addresses and less than 2% false negative cache hits. The false positive and false negative cache hits are due to the high key rate in the keypress simulation.

For verification purposes, we executed the generated keylogger for a period of 60 seconds and randomly pressed keys on the keyboard. In this setting we measured no false positives and no false negatives at all. This results from significantly lower key rates than in the profiling phase. The table is not used by any process other than the spy and the victim process and the probability that the array access happens exactly between the reload and the flush instruction is rather small, as we have longer idle periods than during the profiling phase. Thus, we are able to uniquely identify each key without errors.

Attack on Instruction Executions. The same attack can easily be performed on executed instructions. The source code for this example is shown in Listing 2. Each key is now processed in its own function, as defined by the `CASE(X)` macro. The functions are page aligned to avoid prefetcher activity. The `NOP1024` macro generates 1024 `nop` instructions, which is enough to avoid accidental code prefetching of function code.

Our measurements show that there is no difference between Cache Template Attacks on code and data accesses.

Performance Evaluation. To examine the performance limits of the exploitation phase of Cache Template Attacks, we evaluated the number of addresses which can

```
1 #define NOP1024 /*1024 times asm("nop"); */
2 #define CASE(X) case X: \
3   { ALIGN(0x1000) void f##X() { NOP1024 };\
4     f##X(); break; }
5 int main(int argc, char** argv) {
6   while (1) {
7     int c = getchar(); // unbuffered
8     switch (c) {
9       CASE(0);
10      // ...
11      CASE(128);
12 } } }
```

Listing 2: Victim program with long functions on Linux

be accurately monitored simultaneously at different key rates. At a key rate of 50 keys per second, we managed to spy on 16000 addresses simultaneously on an Intel i5 Sandy Bridge CPU without any false positives or false negatives. The first errors occurred when monitoring 18000 addresses simultaneously. At a key rate of 250 keys per second, which is the maximum on our system, we were able to spy on 4000 addresses simultaneously without any errors. The first errors occurred when monitoring 5000 addresses simultaneously. In both cases, we monitor significantly more addresses than in any practical cache attack today.

However, monitoring that many addresses is only possible if their position in virtual memory is such that the prefetcher remains inactive. Accessing several consecutive addresses on the same page causes prefetching of more data, resulting in cache hits although no program accessed the data. The limiting effect of the prefetcher on the Flush+Reload attack has already been observed by Yarom and Benger [54]. Based on these observations, we discuss the possibility of using the prefetcher as an effective countermeasure against cache attacks in Section 6.3.

5 Attacks on Real-World Applications

In this section, we consider an attack scenario where an attacker is able to execute an attack tool on a targeted machine in unprivileged mode. By executing this attack tool, the attacker extracts the cache-activity profiles which are exploited subsequently. Afterwards, the attacker collects the secret information acquired during the exploitation phase.

For this rather realistic and powerful scenario we present various case studies of attacks launched against real applications. We demonstrate the power of automatically launching cache attacks against any binary or library. First, we launch two attacks on Linux user interfaces, including GDK-based user interfaces, and an attack against a Windows user interface. In all attacks we simulate the user input in the profiling phase. Thus, the attack can be automated on the device under attack. To demonstrate the range of possible applications, we also present an automated attack on the T-table-based AES implementation of OpenSSL 1.0.2 [37].

Attack on Linux User Interfaces

There exists a variety of software-based side-channel attacks on user input data. These attacks either measure differences in the execution time of code in other programs or libraries [48], approximate keypresses through CPU and cache activity [44], or exploit system services leaking user input data [56]. In particular, Zhang et al. [56] use information about other processes from procfs on Linux to measure inter-keystroke timings and derive key sequences. Their proposed countermeasures can be implemented with low costs and prevent their attack completely. We, however, employ Cache Template Attacks to find and exploit leaking side-channel information in shared libraries automatically in order to spy on keyboard input.

Given root access to the system, it is trivial to write a keylogger on Linux using /dev/input/event* devices. Furthermore, the xinput tool can also be used to write a keylogger on Linux, but root access is required to install it. However, using our approach of Cache Template Attacks only requires the unprivileged execution of untrusted code as well as the capability of opening the attacked binaries or shared libraries in a read-only shared memory. In the exploitation phase one round of Flush+Reload on a single address takes less than 100 nanoseconds. If we measure the average latency between keypress and cache hit, we can determine the actual keypress timing up to a few hundred nanoseconds. Compared to the existing attacks mentioned above, our attack is significantly more accurate in terms of both event detection (detection rates near 100%) and timing deviations.

In all attacks presented in this section we compute time-independent cache-hit ratios.

Attack on the GDK Library. Launching the Cache Template profiling phase on different Linux applications revealed thousands of addresses in different libraries, binaries, and data files showing cache activity upon keypresses. Subsequently, we targeted different keypress events in order to find addresses distinguishing the different keys. Figure 3 shows the Cache Template of a memory area in the GDK library libgdk-3.so.0.1000.8, a part of the GTK framework which is the default user-interface framework on many Linux distributions.

Figure 3 shows several addresses that yield a cache hit with a high accuracy if and only if a certain key is

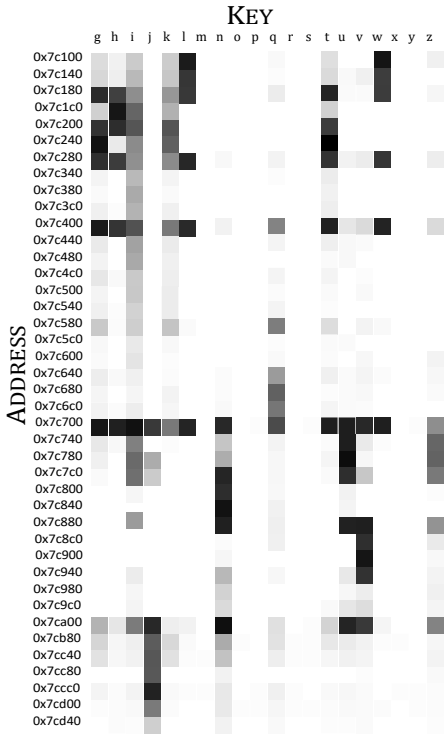

Figure 3: Excerpt of the GDK Cache Template. Dark cells indicate key-address-pairs with high cache-hit ratios.

pressed. For instance, every keypress on key n results in cache hit on address 0x7c800, whereas the same address reacts in only 0.5% of our tests on other keypresses. Furthermore, we found a high cache-hit ratio on some addresses when a key is pressed (i.e., 0x6cd00 in libgdk), the mouse is moved (i.e., 0x28760 in libgdk) or a modifier key is pressed (i.e., 0x72fc0 in libgdk). We also profiled the range of keys a–f but it is omitted from Figure 3 because no high cache-hit ratios have been observed for the shown addresses.

We use the spy tool described in Section 3.2 in order to spy on events based on the Cache Template. We are able to accurately determine the following sets of pressed keys: $\{i\}, \{j\}, \{n\}, \{q\}, \{v\}, \{w\}, \{z\}, \{g, h, k, t\}$. That is, we cannot distinguish between keys in the same set, but keys in one set from keys in other sets. Similarly, we can deduce whether a key is contained in none of these sets.

Not as part of our attack, but in order to understand how keyboard input is processed in the GDK library, we analyzed the binary and the source code. In general, we found out that most of the addresses revealed in the profiling phase point to code executed while processing keyboard input. The address range discussed in this section contains the array gdk_keysym_to_unicode_tab which is used to translate key symbols to unicode special characters. The library performs a binary search on this array, which explains why we can identify certain keys accurately, namely the leaf nodes in the binary search.

As the corresponding array is used for keyboard input in all GDK user-interface components, including password fields, our spy tool works for all applications that use the GDK library. This observation allows us to use Cache Template Attacks to build powerful keyloggers for GDK-based user interfaces automatically. Even if we cannot distinguish all keys from each other, Cache Template Attacks allow us to significantly reduce the complexity of cracking a password. In this scenario, we are able to identify 3 keys reliably, as well as the total number of keypresses. Thus, in case of a lowercase password we can reduce the entropy per character from $\log_2(26) = 4.7$ to 4.0 bits. Attacking more than 3 addresses in order to identify more keys adds a significant amount of noise to the results, as it triggers the prefetcher. First experiments demonstrated the feasibility of attacking the lock screen of Linux distributions. However, further evaluation is necessary in order to reliably determine the effectiveness of this approach.

Attack on GDK Key Remapping. If an attacker has additional knowledge about the attacked system or software, more efficient and more powerful attacks are possible. Inspired by Tannous et al. [48] who performed a timing attack on GDK key remapping, we demonstrate a more powerful attack on the GDK library, by examining how the remapping of keys influences the sets of identifiable keypresses. The remapping functionality uses a large key-translation table gdk_keys_by_keyval which spreads over more than four pages.

Hence, we repeated the Cache Template Attack on the GDK library with a small modification. Before measuring cache activity for an address during an event, we remapped one key to the key code at that address, retrieved from the gdk_keys_by_keyval table. We found significant cache activity for some address and key-remapping combinations.

When profiling each key remapping for $d = 0.8$ seconds, we measured cache activity in 52 cache-line-sized memory regions. In verification scans, we found 0.2-2.5% false positive cache hits in these memory regions. Thus, we have found another highly accurate side channel for specific key remappings. The results are shown in the F-score graph in Figure 4. High values allow accurate detection of keypresses if the key is remapped to this address. Thus, we find more accurate results in terms of timing in our automated attack than Tannous et al. [48].

We can only attack 8 addresses in the profiled memory area simultaneously, since it spreads over 4 pages and we can only monitor 2 or 3 addresses without triggering the prefetcher. Thus, we are able to remap any 8

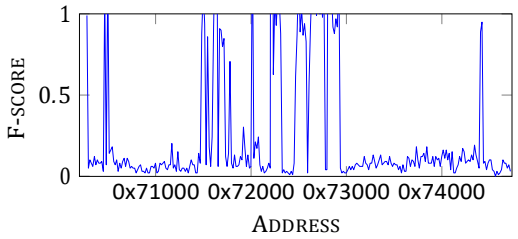

Figure 4: Excerpt of the F-score plot for the address range of the gdk keys_by_keyval table. High values reveal addresses that can be exploited.

keys to these addresses and reliably distinguish them. In combination with the 3 addresses of our previous results, we are able to distinguish at least 11 keys and observe the timestamp of any keystroke in the system based on cache accesses simultaneously.

It is also possible to remap more than one key to the same key code. Hence, it is possible to distinguish between groups of keys. If we consider a lower-case password again, we can now reduce the entropy per character from $\log_2(26) = 4.7$ to 1.4 bits.

We also profiled keypresses on capslock and shift. Although we were able to log keypresses on both keys, we did not consider upper case or mixed case input. The exploitation phase automatically generates a log file containing the information observed through the cache side channel. However, interpretation of these results, such as deriving a program state from a sequence of events (shift key pressed or capslock active) and the influence of the program state on subsequent events is up to analysis of the results after the attack has been performed.

Tannous et al. [48] also described a login-detection mechanism in order to avoid remapping keys unless the user types in a password field. The spy program simply watches /proc to see whether a login program is running. Then the keys are remapped. As soon as the user pauses, the original key mappings are restored. The user will then notice a password mismatch, but the next password entry will work as expected.

Our completely automated password keylogger is a single binary which runs on the attacked system. It maps the GDK library into its own address space and performs the profiling phase. The profiling of each keypress requires the simulation of the keypress into a hidden window. Furthermore, some events require the key remapping we just described. Finally, the keylogger switches into the exploit mode. As soon as a logon screen is detected, for instance, after the screensaver was active or the screen was locked, the keys are remapped and all keypresses are logged into a file accessible by the attacker. Thus, all steps from the deployment of the keylogger to the final log file are fully automated.

Attacks on other Linux Applications

We also found leakage of accurate keypress timings in other libraries, such as the ncurses library (i.e., offset 0xbf90 in libncurses.so), and in files used to cache generated data related to user text input, such as /usr/lib/locale/locale-archive. The latter one is used to translate keypresses into the current locale. It is a generated file which differs on each system and which changes more frequently than the attacked libraries. In consequence, it is not possible to perform an offline attack, i.e., to use a pre-generated Cache Template in the exploitation phase on another system. Still, our concept of Cache Template Attacks allows us to perform an online attack, as profiling is fully automated by generating keystrokes through libxdo or comparable libraries. Thus, keystroke side channels are found within a few seconds of profiling. All keypress-timing side channels we found have a high accuracy and a timing deviation of less than 1 microsecond to the actual keypress.

In order to demonstrate Cache Template Attacks on a low-frequency event which is only indirectly connected to keypresses, we attacked sshd, trying to detect when input is sent over an active ssh connection. The received characters are unrelated to the local user input. When profiling for a duration of $d = 0.8$ seconds per address, we found 428 addresses showing cache activity when a character was received. We verified these results for some addresses manually. None of these checked addresses showed false positive hits within a verification period of 60 seconds. Thus, by exploiting the resulting Cache Template matrix, we are able to gain accurate timings for the transmitted characters (significantly less than 1 microsecond deviation to the transmission of the character). These timings can be used to derive the transmitted letters as shown by Zhang et al. [56].

Attack on Windows User Interfaces

We also performed Cache Template Attacks on Windows applications. The attack works on Windows using MinGW identically to Linux. Even the implementation is the same, except for the keystroke simulation which is now performed using the Windows API instead of the libxdo library, and the file under attack is mapped using LoadLibrary instead of mmap. We performed our attack on Windows 7 and Windows 8.1 systems with the same results on three different platforms, namely Intel Core 2 Duo, Intel i5 Sandy Bridge, and Intel i5 Ivy Bridge. As in the attacks on Linux user interfaces, address space layout randomization has been activated during both profiling and exploitation phase.

In an automated attack, we found cache activity upon keypresses in different libraries with reasonable accu-

racy. For instance, the Windows 7 common control library comctl32.dll can be used to detect keypresses on different addresses. Probing 0xc5c40 results in cache hits on every keypress and mouse click within text fields accurately. Running the generated keypress logger in a verification period of 60 seconds with keyboard input by a real user, we found only a single false positive event detection based on this address. Address 0xc6c00 reacts only on keypresses and not on mouse clicks, but yields more false positive cache hits in general. Again, we can apply the attack proposed by Zhang et al. [56] to recover typed words from inter-keystroke timings.

We did not disassemble the shared library and therefore do not know which function or data accesses cause the cache hit. The addresses were found by starting the Cache Template Attack with the same parameters as on Linux, but on a Windows shared library instead of a Linux shared library. As modern operating systems like Windows 7 and Windows 8.1 employ an immense number of shared libraries, we profiled only a few of these libraries. Hence, further investigations might even reveal addresses for a more accurate identification of keypresses.

Attack on a T-table-based AES

Cache attacks have been shown to enable powerful attacks against cryptographic implementations. Thus, appropriate countermeasures have already been suggested for the case of AES [15, 25, 30, 43]. Nevertheless, in order to compare the presented approach of Cache Template Attacks to related attacks, we launched an efficient and automated access-driven attack against the AES T-table implementation of OpenSSL 1.0.2, which is known to be insecure and susceptible to cache attacks [2, 4, 5, 16, 21, 22, 39, 53]. Recall that the T-tables are accessed according to the plaintext p and the secret key k, i.e., $T_j[p_i \oplus k_i]$ with $i \equiv j \mod 4$ and $0 \leq i < 16$, during the first round of the AES encryption. For the sake of brevity, we omit the full details of an access-driven cache attack against AES and refer the interested reader to the work of Osvik et al. [39, 49].

Attack of Encryption Events. In a first step, we profiled the two events "no encryption" and "encryption with random key and random plaintext". We profiled each cache-line-aligned address in the OpenSSL library during 100 encryptions. On our test system, one encryption takes around 320 cycles, which is very fast compared to a latency of at least 200 cycles caused by a single cache miss. In order to make the results more deterministically reproducible, we measure whether a cache line was used only after the encryption has finished. Thus, the profiling phase does not run in parallel and only one cache hit or miss is measured per triggered event.

This profiling step takes less than 200 seconds. We detected cache activity on 0.2%-0.3% of the addresses. Only 82 addresses showed a significant difference in cache activity depending on the event. For 18 of these addresses, the cache-hit ratio was 100% for the encryption event. Thus, our generated spy tool is able to accurately detect whenever an encryption is performed.

For the remaining 64 addresses the cache-hit ratio was around 92% for the encryption event. Thus, not each of these addresses is accessed in every encryption, depending on key and plaintext. Since we attack a T-table-based AES implementation, we know that these 64 addresses must be the T-tables, which occupy 4 KB respectively 64 cache lines. Although this information is not used in the first generated spy tool, it encourages performing a second attack to target specific key-byte values.

Attack on Specific Key-Byte Values. Exploiting the knowledge that we attack a T-table implementation, we enhance the attack by profiling over different key-byte values for a fixed plaintext, i.e., the set of events consists of the different key-byte values. Our attack remains fully automated, as we change only the values with which the encryption is performed. The result is again a log file containing the accurate timestamp of each event monitored. The interpretation of the log file, of course, involves manual work and is specific to the targeted events, i.e., key bytes in this case.

For each key byte k_i, we profile only the upper 4 bits of k_i as the lower 4 bits cannot be distinguished because of the cache-line size of 64 bytes. This means that we need to profile only 16 addresses for each key byte k_i. Furthermore, on average 92% of these addresses are already in the cache and the Reload step of the Flush+Reload attack is unlikely to trigger the prefetcher. Thus, we can probe all addresses after a single encryption. Two profiles for different values of k_0 are shown in Figure 5. The two traces were generated using 1000 encryptions per key byte and address to show the pattern more clearly. According to Osvik et al. [39] and Spreitzer et al. [46] these plots (or patterns) reveal at least the upper 4 bits of a key byte and, hence, attacking the AES T-table implementation works as expected. In our case, experiments showed that 1 to 10 encryptions per key byte are enough to infer these upper 4 bits correctly.

In a T-table-based AES implementation, the index of the T-table is determined by $p_i \oplus k_i$. Therefore, the same profiles can be generated by iterating over the different plaintext byte values while encrypting with a fixed key. Osvik et al. [39] show a similar plot, generated using the Evict+Time attack. However, in our attack the profiles are aggregated into the Cache Template matrix, as de-

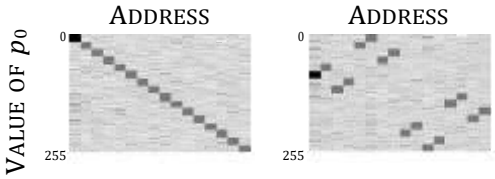

Figure 5: Excerpt of the Cache Template (address range of the first T-table). The plot is transposed to match [39]. In the left trace $k_0 = 0x00$, in the right trace $k_0 = 0x51$.

scribed in Section 3.1.

In the exploitation phase, the automatically generated spy tool monitors cache hits on the addresses from the Cache Template in order to determine secret key-byte values. We perform encryptions using chosen plaintexts. We attack the 16 key bytes k_i sequentially. In each step $i = 0, \ldots, 15$, the plaintext is random, except for the upper 4 bits of p_i, which are fixed to the same chosen value as in the profiling phase. Hence, the encryption is performed over a chosen plaintext. The spy tool triggers an encryption, detects when the encryption actually happens and after each encryption, reports the set of possible values for the upper 4 bits of key byte k_i. As soon as only one candidate for the upper 4 bits of key byte k_i remains, we continue with the next key byte.

Using Cache Template Attacks, we are able to infer 64 bits of the secret key with only 16–160 encryptions in a chosen-plaintext attack. Compared to the work of Osvik et al. [39] who require several hundred or thousands encryptions (depending on the measurement approach) targeting the L1 cache, and the work of Spreitzer and Plos [46] who require millions of encryptions targeting the L1 cache on the ARM platform, we clearly observe a significant performance improvement. More recent work shows that full key recovery is possible with less than 30000 encryptions [17] using Flush+Reload.

The benefit of our approach, compared to existing cache attacks against AES, is that our attack is fully automated. Once the binary is deployed on the target system, it performs both profiling and exploitation phase automatically and finally returns a log file containing the key byte candidates to the attacker. Moreover, we do not need prior knowledge of the attacked system or the attacked executable or library.

AES T-table implementations are already known to be insecure and countermeasures have already been integrated, e.g., in the AES implementation of OpenSSL. Performing our attack on a non-T-table implementation (e.g., by employing AES-NI instructions) did not show key dependent information leakage, but still, we can accurately determine the start and end of the encryption through the cache behavior. However, we leave it as an interesting open issue to employ the presented approach of cache template attacks for further investigations of vulnerabilities in already protected implementations.

Trace-Driven Attack on AES. When attacking an insecure implementation of a cryptographic algorithm, an attacker can often gain significantly more information if it is possible to perform measurements during the encryption [2, 13], i.e., in case the exact trace of cache hits and cache misses can be observed. Even if we cannot increase the frequency of the Flush+Reload attack, we are able to slow down the encryption by constantly flushing the 18 addresses which showed cache activity in every profile. We managed to increase the encryption time from 320 cycles to 16000–20000 cycles. Thus, a more fine-grained trace of cache hits and cache misses can be obtained which might even allow the implementation of trace-driven cache attacks purely in software.

6 Countermeasures

We have demonstrated in Section 5 that Cache Template Attacks are applicable to real-world applications without knowledge of the system or the application. Therefore, we emphasize the need for research on effective countermeasures against cache attacks. In Section 6.1, we discuss several countermeasures which have been proposed so far. Subsequently, in Section 6.2, we discuss how Cache Template Attacks can be employed by developers to detect and eliminate cache-based information leakage and also by users to detect and prevent cache attacks running actively on a system. Finally, in Section 6.3, we propose changes to the prefetcher to build a powerful countermeasure against cache attacks.

Discussion of Countermeasures

Removal of the clflush Instruction is not Effective. The restriction of the clflush instruction has been suggested as a possible countermeasure against cache attacks in [54, 55, 58]. However, by adapting our spy tool to evict the cache line without using the clflush instruction (Evict+Reload instead of Flush+Reload), we demonstrate that this countermeasure is not effective at all. Thereby, we show that cache attacks can be launched successfully even without the clflush instruction.

Instead of using the clflush instruction, the eviction is done by accessing physically congruent addresses in a large array which is placed in large pages by the operating system. In order to compute physically congruent addresses we need to determine the lowest 18 bits of the physical address to attack, which can then be used to evict specific cache sets.

The actual mapping of virtual to physical addresses can be retrieved from /proc/self/pagemap. Even if

such a mapping is not available, methods to find congruent addresses have been developed—simultaneously to this work—by Irazoqui et al. [20] by exploiting large pages, Oren et al. [38] by exploiting timing differences in JavaScript, and Liu et al. [32] by exploiting timing differences in native code.

The removal of the clflush instruction has also been discussed as a countermeasure to protect against DRAM disturbance errors (denoted as rowhammer bug). These disturbance errors have been studied by Kim et al. [27] and, later on, exploited by Seaborn et al. [45] to gain kernel privileges. Several researchers have already claimed to be able to exploit the rowhammer bug without the clflush instruction [14], This can be done by exploiting the Sandy Bridge cache mapping function, which has been reverse engineered by Hund et al. [18], to find congruent addresses.

Our eviction strategy only uses the lowest 18 bits and therefore, we need more than 12 accesses to evict a cache line. With 48 accessed addresses, we measured an eviction rate close to 100%. For performance reasons we use write accesses, as the CPU does not have to wait for data fetches from the physical memory. In contrast to the clflush instruction, which takes only 41 cycles, our eviction function takes 325 cycles. This is still fast enough for most Flush+Reload attacks.

While clflush always evicts the cache line, our eviction rate is only near 100%. Therefore, false positive cache hits occur if the line has not been evicted. Using Flush+Reload, there is a rather low probability for a memory access on the monitored address to happen exactly between the Reload step and the point where the clflush takes effect. This probability is much higher in the case of Evict+Reload, as the eviction step takes 8 times longer than the clflush instruction.

We compare the accuracy of Evict+Reload to Flush+Reload using previously found cache vulnerabilities. For instance, as described in Section 5.1, probing address 0x7c800 of libgdk-3.so.0.1000.8 allows us to detect keypresses on key n. The Flush+Reload spy tool detects on average 98% of the keypresses on key n with a 2% false positive rate (keypresses on other keys). Using Evict+Reload, we still detect 90% of the keypresses on key n with a 5% false positive rate. This clearly shows that the restriction of clflush is not sufficient to prevent this type of cache attack.

Disable Cache-Line Sharing. One prerequisite of Flush+Reload attacks is shared memory. In cloud scenarios, shared memory across virtual machine borders is established through page deduplication. Page deduplication between virtual machines is commonly disabled in order to prevent more coarse-grained attacks like fingerprinting operating systems and files [40, 47] as well as Flush+Reload. Still, as shown by Irazoqui et al. [20], it is possible to use Prime+Probe as a fallback. However, attacking low-frequency events like keypresses becomes infeasible, because Prime+Probe is significantly more susceptible to noise.

Flush+Reload can also be prevented on a system by preventing cache-line sharing, i.e., by disabling shared memory. Unfortunately, operating systems make heavy use of shared memory, and without modifying the operating system it is not possible for a user program to prevent its own memory from being shared with an attacker, even in the case of static linkage as discussed in Section 2.2.

With operating-system modifications, it would be possible to disable shared memory in all cases where a victim program cannot prevent an attack, i.e., shared program binaries, shared libraries, shared generated files (for instance, locale-archive). Furthermore, it would be possible to provide a system call to user programs to mark memory as "do-not-share."

A hardware-based approach is to change cache tags. Virtually tagged caches are either invalidated on context switches or the virtual tag is combined with an address space identifier. Therefore, shared memory is not shared in the cache. Thus, Flush+Reload is not possible on virtually tagged caches.

We emphasize that as long as shared cache lines are available to an attacker, Flush+Reload or Evict+Reload cannot be prevented completely.

Cache Set Associativity. Prime+Probe, Evict+Time and Evict+Reload exploit set-associative caches. In all three cases, it is necessary to fill all ways of a cache set, either for eviction or for the detection of evicted cache sets. Based on which cache set was reloaded (respectively evicted), secret information is deduced. Fully associative caches have better security properties, as such information deduction is not possible and cache eviction can only be enforced by filling the whole cache. However, a timing attack would still be possible, e.g., due to internal cache collisions [5] leading to different execution times. As fully associative caches are impractical for larger caches, new cache architectures have been proposed to provide similar security properties [29, 51, 52]. However, even fully associative caches only prevent attacks which do not exploit cache-line sharing. Thus, a combination of countermeasures is necessary to prevent most types of cache attacks.

Proactive Prevention of Cache Attacks

Instrumenting cache attacks to detect co-residency [57] with another virtual machine on the same physical machine, or even to detect cache attacks [58] and cache-based side channels in general [11] has already been pro-

posed in the past. Moreover, Brumley and Hakala [7] even suggested that developers should use their attack technique to detect and eliminate cache vulnerabilities in their programs. Inspired by these works, we present defense mechanisms against cache attacks which can be improved by using Cache Template Attacks.

Detect Cache Vulnerabilities as a Developer. Similar to Brumley and Hakala [7], we propose the employment of Cache Template Attacks to find cache-based vulnerabilities automatically. Compared to [7], Cache Template Attacks allow developers to detect potential cache side channels for specifically chosen events automatically, which can subsequently be fixed by the developer. A developer only needs to select the targeted events (e.g., keystrokes, window switches, or encryptions) and to trigger these events automatically during the profiling phase, which significantly eases the evaluation of cache side channels. Ultimately, our approach even allows developers to find such cache vulnerabilities in third party libraries.

Detect and Impede Ongoing Attacks as a User. Zhang et al. [58] stated the possibility to detect cache attacks by performing a cache attack on one of the vulnerable addresses or cache sets. We propose running a Cache Template Attack as a system service to detect code and data under attack. If Flush+Reload prevention is sufficient, we simply disable page sharing for all pages with cache lines under attack. Otherwise, we disable caching for these pages as proposed by Aciiçmez et al. [1] and, thus, prevent all cache attacks. Only the performance for critical code and data parts is reduced, as the cache is only disabled for specific pages in virtual memory.

Furthermore, cache attacks can be impeded by performing additional memory accesses, unrelated to the secret information, or random cache flushes. Such obfuscation methods on the attacker's measurements have already been proposed by Zhang et al. [59]. The idea of the proposed obfuscation technique is to generate random memory accesses, denoted as cache cleansing. However, it does not address the shared last-level cache. In contrast, Cache Template Attacks can be used to identify possible cache-based information leaks and then to specifically add noise to these specific locations by accessing or flushing the corresponding cache lines.

Enhancing the Prefetcher

During our experiments, we found that the prefetcher influences the cache activity of certain access patterns during cache attacks, especially due to the spatial locality of addresses, as also observed in other work [16, 39, 54]. However, we want to discuss the prefetcher in more detail as it is crucial for the success of a cache attack.

Although the profiling phase of Cache Template Attacks is not restricted by the prefetcher, the spy program performing the exploitation phase might be unable to probe all leaking addresses simultaneously. For instance, we found 255 addresses leaking side-channel information about keypresses in the GDK library but we were only able to probe 8 of them simultaneously in the exploitation phase, because the prefetcher loads multiple cache lines in advance and, thus, generates numerous false positive cache hits.

According to the Intel 64 and IA-32 Architectures Optimization Reference Manual [19], the prefetcher loads multiple memory addresses in advance if "two cache misses occur in the last level cache" and the corresponding memory accesses are within a specific range (the so-called trigger distance). Depending on the CPU model this range is either 256 or 512 bytes, but does not exceed a page boundary of 4 KB. Due to this, we are able to probe at least 2 addresses per page.

We suggest increasing the trigger distance of the prefetcher beyond the 4 KB page boundary if the corresponding page already exists in the translation lookaside buffer. The granularity of the attack will then be too high for many practical targets, especially attacks on executed instructions will then be prevented.

As cache attacks constantly reaccess specific memory locations, another suggestion is to adapt the prefetcher to take temporal spatiality into consideration. If the prefetcher were to prefetch data based on that temporal distance, most existing attacks would be prevented.

Just as we did in Section 4, an attacker might still be able to establish a communication channel targeted to circumvent the prefetcher. However, the presented countermeasures would prevent most cache attacks targeting real-world applications.

7 Conclusion

In this paper, we introduced Cache Template Attacks, a novel technique to find and exploit cache-based side channels easily. Although specific knowledge of the attacked machine and executed programs or libraries helps, it is not required for a successful attack. The attack is performed on closed-source and open-source binaries in exactly the same way.

We studied various applications of Cache Template Attacks. Our results show that an attacker is able to infer highly accurate keystroke timings on Linux as well as Windows. For Linux distributions we even demonstrated a fully automatic keylogger that significantly reduces the entropy of passwords. Hence, we conclude that cache-based side-channel attacks are an even greater threat for

today's computer architectures than assumed so far. In fact, even sensitive user input, like passwords, cannot be considered secure on machines employing CPU caches.

We argue that fundamental concepts of computer architectures and operating systems enable the automatic exploitation of cache-based vulnerabilities. We observed that many of the existing countermeasures do not prevent such attacks as expected. Still, the combination of multiple countermeasures can effectively mitigate cache attacks. However, the fact that cache attacks can be launched automatically marks a change of perspective, from a more academic interest towards practical attacks, which can be launched by less sophisticated attackers. This shift emphasizes the need to develop and integrate effective countermeasures immediately. In particular, it is not sufficient to protect only specific cryptographic algorithms like AES. More general countermeasures will be necessary to counter the threat of automated cache attacks.

8 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Ben Ransford, for their valuable comments and suggestions.

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644052 (HECTOR).

Furthermore, this work has been supported by the Austrian Research Promotion Agency (FFG) and the Styrian Business Promotion Agency (SFG) under grant number 836628 (SeCoS).

References

[1] Aciiçmez, O., Brumley, B. B., and Grabher, P. New Results on Instruction Cache Attacks. In *Cryptographic Hardware and Embedded Systems – CHES* (2010), vol. 6225 of *LNCS*, Springer, pp. 110–124.

[2] Aciiçmez, O., and Koç, Ç. K. Trace-Driven Cache Attacks on AES (Short Paper). In *International Conference on Information and Communications Security – ICICS* (2006), vol. 4307 of *LNCS*, Springer, pp. 112–121.

[3] Benger, N., van de Pol, J., Smart, N. P., and Yarom, Y. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In *Cryptographic Hardware and Embedded Systems – CHES* (2014), vol. 8731 of *LNCS*, Springer, pp. 75–92.

[4] Bernstein, D. J. Cache-Timing Attacks on AES, 2004. URL: http://cr.yp.to/papers.html#cachetiming.

[5] Bogdanov, A., Eisenbarth, T., Paar, C., and Wienecke, M. Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs. In *Topics in Cryptology – CT-RSA* (2010), vol. 5985 of *LNCS*, Springer, pp. 235–251.

[6] Bonneau, J., and Mironov, I. Cache-Collision Timing Attacks Against AES. In *Cryptographic Hardware and Embedded Systems – CHES* (2006), vol. 4249 of *LNCS*, Springer, pp. 201–215.

[7] Brumley, B. B., and Hakala, R. M. Cache-Timing Template Attacks. In *Advances in Cryptology – ASIACRYPT* (2009), vol. 5912 of *LNCS*, Springer, pp. 667–684.

[8] Chari, S., Rao, J. R., and Rohatgi, P. Template Attacks. In *Cryptographic Hardware and Embedded Systems – CHES* (2002), vol. 2523 of *LNCS*, Springer, pp. 13–28.

[9] Chen, C., Wang, T., Kou, Y., Chen, X., and Li, X. Improvement of Trace-Driven I-Cache Timing Attack on the RSA Algorithm. *Journal of Systems and Software 86*, 1 (2013), 100–107.

[10] Daemen, J., and Rijmen, V. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[11] Doychev, G., Feld, D., Köpf, B., Mauborgne, L., and Reineke, J. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium* (2013), USENIX Association, pp. 431–446.

[12] Franz, M. E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism. In *Workshop on New Security Paradigms – NSPW* (2010), ACM, pp. 7–16.

[13] Gallais, J., Kizhvatov, I., and Tunstall, M. Improved Trace-Driven Cache-Collision Attacks against Embedded AES Implementations. *IACR Cryptology ePrint Archive 2010/408*.

[14] Google Groups. Rowhammer without CLFLUSH, 2015. URL: https://groups.google.com/forum/#!topic/rowhammer-discuss/ojgTgLr4q_M.

[15] Gueron, S. White Paper: Intel Advanced Encryption Standard (AES) Instructions Set, 2010. URL: https://software.intel.com/file/24917.

[16] Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy – S&P* (2011), IEEE Computer Society, pp. 490–505.

[17] Gülmezoğlu, B., Inci, M. S., Eisenbarth, T., and Sunar, B. A Faster and More Realistic Flush+Reload Attack on AES. In *Constructive Side-Channel Analysis and Secure Design – COSADE* (2015), LNCS, Springer. In press.

[18] Hund, R., Willems, C., and Holz, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy – SP* (2013), IEEE Computer Society, pp. 191–205.

[19] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. No. 248966-026. 2012.

[20] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *IEEE Symposium on Security and Privacy – S&P* (2015), IEEE Computer Society.

[21] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Fine grain Cross-VM Attacks on Xen and VMware are possible! *IACR Cryptology ePrint Archive 2014/248*.

[22] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses Symposium – RAID* (2014), vol. 8688 of *LNCS*, Springer, pp. 299–319.

[23] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Know Thy Neighbor: Crypto Library Detection in Cloud. *Privacy Enhancing Technologies 1*, 1 (2015), 25–40.

[24] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Lucky 13 Strikes Back. In *ACM ASIA CCS* (2015), pp. 85–96.

[25] Käsper, E., and Schwabe, P. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems – CHES* (2009), vol. 5747 of *LNCS*, Springer, pp. 1–17.

[26] Kelsey, J., Schneier, B., Wagner, D., and Hall, C. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security 8*, 2/3 (2000), 141–158.

[27] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ACM/IEEE International Symposium on Computer Architecture – ISCA* (2014), IEEE Computer Society, pp. 361–372.

[28] Kocher, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO* (1996), vol. 1109 of *LNCS*, Springer, pp. 104–113.

[29] Kong, J., Aciiçmez, O., Seifert, J., and Zhou, H. Deconstructing New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *ACM Workshop on Computer Security Architecture – CSAW* (2008), pp. 25–34.

[30] Könighofer, R. A Fast and Cache-Timing Resistant Implementation of the AES. In *Topics in Cryptology – CT-RSA* (2008), vol. 4964 of *LNCS*, Springer, pp. 187–202.

[31] Liu, F., and Lee, R. B. Random Fill Cache Architecture. In *International Symposium on Microarchitecture – MICRO* (2014), IEEE, pp. 203–215.

[32] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy – S&P* (2015).

[33] Maurice, C., Neumann, C., Heen, O., and Francillon, A. C5: Cross-Cores Cache Covert Channel. In *DIMVA* (2015). In press.

[34] Mowery, K., Keelveedhi, S., and Shacham, H. Are AES x86 Cache Timing Attacks Still Feasible? In *Workshop on Cloud Computing Security – CCSW* (2012), ACM, pp. 19–24.

[35] National Institute of Standards and Technology. Advanced Encryption Standard. NIST FIPS PUB 197, 2001.

[36] Neve, M. *Cache-based Vulnerabilities and SPAM Analysis*. PhD thesis, UCL, 2006.

[37] OpenSSL Software Foundation. OpenSSL Project, 2014. URL: http://www.openssl.org/.

[38] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The Spy in the Sandbox - Practical Cache Attacks in Javascript. *CoRR abs/1502.07373* (2015).

[39] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA* (2006), vol. 3860 of *LNCS*, Springer, pp. 1–20.

[40] Owens, R., and Wang, W. Non-Interactive OS Fingerprinting Through Memory De-Duplication Technique in Virtual Machines. In *International Performance Computing and Communications Conference – IPCCC* (2011), IEEE, pp. 1–8.

[41] Page, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptology ePrint Archive 2002/169*.

[42] Percival, C. Cache Missing for Fun and Profit, 2005. URL: http://www.daemonology.net/hyperthreading-considered-harmful/.

[43] Rebeiro, C., Selvakumar, A. D., and Devi, A. S. L. Bitslice Implementation of AES. In *Cryptology and Network Security – CANS* (2006), vol. 4301 of *LNCS*, Springer, pp. 203–212.

[44] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *ACM Conference on Computer and Communications Security – CCS* (2009), ACM, pp. 199–212.

[45] Seaborn, M., and Dullien, T. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges, 2015. URL: http://googleprojectzero.blogspot.co.at/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[46] Spreitzer, R., and Plos, T. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *Constructive Side-Channel Analysis and Secure Design – COSADE* (2013), vol. 7864 of *LNCS*, Springer, pp. 200–214.

[47] Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Memory Deduplication as a Threat to the Guest OS. In *European Workshop on System Security – EUROSEC* (2011), ACM, pp. 1–6.

[48] Tannous, A., Trostle, J. T., Hassan, M., McLaughlin, S. E., and Jaeger, T. New Side Channels Targeted at Passwords. In *Annual Computer Security Applications Conference – ACSAC* (2008), pp. 45–54.

[49] Tromer, E., Osvik, D. A., and Shamir, A. Efficient Cache Attacks on AES, and Countermeasures. *Journal Cryptology 23*, 1 (2010), 37–71.

[50] Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., and Miyauchi, H. Cryptanalysis of DES Implemented on Computers with Cache. In *Cryptographic Hardware and Embedded Systems – CHES* (2003), vol. 2779 of *LNCS*, Springer, pp. 62–76.

[51] Wang, Z., and Lee, R. B. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture – ISCA* (2007), pp. 494–505.

[52] Wang, Z., and Lee, R. B. A Novel Cache Architecture with Enhanced Performance and Security. In *IEEE/ACM International Symposium on Microarchitecture – MICRO* (2008), pp. 83–93.

[53] Weiss, M., Heinz, B., and Stumpf, F. A Cache Timing Attack on AES in Virtualization Environments. In *Financial Cryptography and Data Security – FC* (2012), vol. 7397 of *LNCS*, Springer, pp. 314–328.

[54] Yarom, Y., and Benger, N. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive 2014/140*.

[55] Yarom, Y., and Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014), USENIX Association, pp. 719–732.

[56] Zhang, K., and Wang, X. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security Symposium* (2009), USENIX Association, pp. 17–32.

[57] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *IEEE Symposium on Security and Privacy – S&P* (2011), IEEE Computer Society, pp. 313–328.

[58] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *ACM Conference on Computer and Communications Security – CCS* (2014), ACM, pp. 990–1003.

[59] Zhang, Y., and Reiter, M. K. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *ACM Conference on Computer and Communications Security – CCS* (2013), ACM, pp. 827–838.