MASCAT: Stopping Microarchitectural Attacks Before Execution

Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar

Worcester Polytechnic Institute, Worcester, MA, USA
{girazoki,teisenbarth,sunar}@wpi.edu

Abstract. Microarchitectural attacks have gained popularity in recent years since they use only standard resources, e.g. memory and cache access timing. Such privileges are available to applications at the lowest privilege levels. Further, microarchitectural attacks have proven successful on shared cloud instances across VMs, on smartphones with sandboxing, and on numerous embedded platforms. Given the rise of malicious code in app stores and in online repositories it becomes essential to scan applications for such stealthy attacks. We present a static code analysis tool, MASCAT , capable of scanning for ever evolving microarchitectural attacks. Our proposed tool MASCAT can be used by app store service providers to perform large scale fully automated analysis of applications. The initial MASCAT suite is built to include attack vectors to cover popular cache/DRAM access attacks and Rowhammer. Further, our tool is easily extensible to cover newer attack vectors as they emerge.

Keywords: Microarchitectural attacks, cache attacks, static code analysis.

1 Introduction

In recent years the security community has witnessed the rise of microachitectural side channel attacks. While only 3 years ago the feasibility of microarchitectural attacks was restricted to core private resources, today they are applicable across CPU cores and even across CPU sockets. Furthermore, new sophisticated methodologies are discovered frequently, e.g. memory bus locking attacks or rowhammer attacks, raising concerns among software and hardware security developers.

In fact, microarchitectural attacks already provide a wide range of their capabilities. In the last few years we have witnessed cache attacks recovering cryptographic keys [20, 18, 40, 5, 9], recover aggregate sensitive information, e.g. the number of items in a shopping cart [44], or enable spying on a user, i.e. through recovery of keystrokes [12] and private messages in a TLS session [21]. In addition, memory bus locking attacks have shown to be capable of acting as covert-channels, detecting hardware co-residency and working as a Quality of Service (QoS) degradation tools [35, 17]. Lastly, rowhammer attacks pushed the envelope further by introducing cryptographic faults or by breaking memory

isolation techniques [7, 37]. What makes microarchitectural attacks so powerful, is not only the capability they provide, but also wide range of situations they apply. Contrary to early popular belief, microarchitectural attacks have proven to work in commercial clouds (e.g. Amazon EC2) [15], in browsers as Javascript extensions [27], in trusted execution environments or even on mobile devices, e.g. as smartphone applications [25]. In short, microarchitectural attacks are applicable and even practical in numerous security-critical scenarios how we use technology every day.

Because of the threat posed it is important that applications distributed by official digital application stores, e.g., Microsoft store or Android Google Play, ensure the sanity of the binaries they release. In fact applications being released by these distributors have the full trust of the customers. However, unlike other kind of malware, microarchitectural attacks are harder to detect as their representation does not look harmful. A common solution is to utilize an anti-virus software to detect malicious code inside binaries. Nevertheless, as it will be presented later in this document, these mis-detect the entire range of microarchitectural attacks, as they are based on (apparently) innocent instructions.

Other solutions have been proposed to protect against microarchitectural attacks, which may be divided into preventive and reactive (online detection) categories. Preventive countermeasures can be adopted at three layers: application layer, OS/hypervisor design layer or at the hardware design layer. However, none of these mechanisms help official application distributors to avoid the release of microarchitectural attack binaries. The implementation of preventive methodologies (e.g., page coloring or Intel Cache Allocation Technology [26]) is not in control of distributors, and indeed are rarely found in common devices. The reactive approaches, such as the monitoring of performance counter events by the OS [42] or the execution of the potentially malicious code inside a secure environment, also fail on the detection of microarchitectural attack code. The first again depends on the behavior of an OS not in control of the application distributor, while the latter can be bypassed by designing code that executes benign code until one date, and malicious after. Thus, official application distributors lack of a solution to prevent the release of malicious microarchitectural attack code, but yet they are the ones to blame if one of those malicious applications succeeds.

Our Contribution

To cope with the aforementioned issues, we present MASCAT (Micro-Architectural Side Channel Attack Trapper), a tool to detect microarchitectural attacks through static analysis of binaries. In short MASCAT is similar to an antivirus tool to catch microarchitectural attacks embedded in innocent looking software. MASCAT works by statically analyzing binary elf files looking for implicit characteristics that microarchitectural attacks usually exhibit in their design. Our approach is divided in several characteristics whose importance factor can be configurable to trigger all or a specific subset of microarchitectural attacks. Further, our approach is designed to easily add more characteristics to the range of attacks that

MASCAT already covers. The execution of our approach is fully automated and in comparison to other solutions, the outcome is easily understandable as it not only colors the location where the threat was found in the binary but also adds an explanation on the characteristics that generated the threat.

Due to its similarity to antivirus tools, MASCAT can be adopted by digital application distributors (like Androids GooglePlay, Microsoft Store or Apple app store) to ensure the applications being offered do not contain microarchitectural attacks. In fact they can use MASCAT to detect the presence of microarchitectural attack characteristics, and decide whether the application is dropped (never placed in the store) or modified (if the source code is available). More than that, if such an attack is found, the submitter can be banned from submitting any other application. Further, MASCAT can be easily adopted by other users that are already used to manage antivirus software, and that would find technically challenging to adopt any of the other existing countermeasures.
In summary, this work

- shows for the first time that app store and end users can stop microarchitectural attacks prior to their execution without the collaboration of OS/software designers;
- introduces a novel a static binary analysis approach looking for attributes implicit to microarchitectural attacks in apparently innocent binaries;
- implements a configurable threat score based approach that is not only easily changeable but also expandable;
- presents a full analysis of 25 attack codes designed by different research groups together with an analysis on 100 benchmarking tools. Our results show a maximum of 6% false positives and a minimum of 0% false negatives;

After reviewing additional related work in Section 2 we discuss existing cache-based side-channel attacks in 3 and their implicit characteristics in Section 4. The proposed tool is introduced in Section 5. The experimental setup and results are presented in Section 6 and 7. We present the conclusions in Section 8.

2 Related Work

Microarchitectural attacks were first theoretically studied in 1992 by Hu, whose analysis was later expanded considering cache hits and misses by Tsunoo et al. in 2003. The first practical microarchitectural attacks (in the form of L1 data attacks) were proposed by Bernstein [6] and Osvik et al. [28]. The first one proposed an attack based on profiling L1 cache collisions while the latter proposed two novel spy process techniques (including the well-known `Prime and Probe` technique) to create cache contentions. Shortly later, Aciimez et al. [2] demonstrated that the spy process techniques were also applicable in the L1 instruction cache by creating set-colliding instructions. Again Aciimez et al. [3] showed that Branch Prediction Units give a similar leakage by inferring whether a victim branch was taken.

In 2009, Ristenpart et al. [32] demonstrated that microarchitectural attacks are applicable in commerical clouds by recovering key strokes from core co-resident VMs. Shortly later, Zhang et al. [45] performed the first fine grain attack across VMs by recovering an El Gamal decryption key from a core co-resident VM. At the same time, Gullasch et al. [13] showed that an AES key can be obtained with very few microarchitectural attack traces if an attacker gains control over the Control Fair Scheduler.

It was in 2013 when Yarom et al. [40] presented the Flush and Reload attack, capable of recovering a RSA key across VMs located in different cores. Shortly later Irazoqui et al. [20] showed that the same technique can be applied to recover AES keys. The Flush and Reload attack was demonstrated to succeed in many other scenarion, like PaaS clouds [44], as a method to perform cache template attacks [12], recover TLS messages [21], work across CPU sockets [19] or even work across smartphone applications [25].

One of the main downsides of the Flush and Reload attack is that it requires memory deduplication to succeed. In 2015, concurrent works from Liu et al. [9] and Irazoqui et al. [18] demonstrated to bypass this requirement by implementing the Prime and Probe attack on the LLC. Later, Oren et al. [27] showed that this attack can be further executed from a javascript inside a local browser, Inci et al. [16] demonstrated its feasibility in commerical clouds and Gruss et al. presented its applicability across cores and in mobile devices [29, 25]. Pessl et al. used similar techniques to perform a cross-core DRAM access based attack [30].

In contrast, rowhammer attacks have been exploited for as little as two years. In 2014, Kim et al. [23] discovered that by constantly accessing a DRAM location bit flips can be induced in adjacent DRAM rows, and called this effect the rowhammer attack. Shortly later, Gruss et al. [11] demonstrated that one can implement rowhammer attacks from javascript, while Bhattacharya et al. [7] showed that faults in cryptographic implementations can be induced with the same technique. Recently, Xiao et al. [37] demonstrated that intelligent execution of rowhammer attacks from a VM to break the memory isolation provided by virtualization, while Van der veen et al. [34] showed the applicability of rowhammer in mobile platforms.

Memory bus locking attacks were presented by Varadarajan et al. [35] and Xu et al. [38] as a mechanism to detect performance degradations and infer co-residency. Shortly later Zhang et al. [43] and Inci et al. [17] demonstrated that the technique can be used to cause Quality of Service degradation in co-resident processes.

Microarchitectural attack defenses have also been widely studied at several levels. In fact, microarchitectural attacks against security critical software exploit human coding mistakes. Aiming at fixing those mistakes, code sanity verification frameworks were proposed in [4] and [41]. However, microarchitectural attacks can also be stopped at the Operating System level without relying on the correctness of the software being executed. For instance, Taesoo et al. [22] proposed a page coloring mechanism to avoid cache attack collisions, while Liu et al. [26] utilized the Intel Cache Allocation Technology (CAT) to prevent cache

attacks. Similarly Zhou et al. [46] propose new page access based countermeasures against cache side channel attacks and Basser et al. [8] propose two software based mechanisms to prevent rowhammer attacks. Finally, countermeasures also can be adopted at the hardware level, e.g. by proposing new microarchitectural isolation techniques to avoid exloitations [36].

3 Microarchitectural Attacks

Microarchitectural attacks take advantage of shared hardware contention to infer information from a co-resident victim. The choice of the hardware piece to target usually becomes a trade-off between feasibility and threat exposure. In the last years several practical microarchitectural attacks with severe privacy/availability violation implications have been proposed, most of which we include in our static analysis approach. This section gives a brief description on the functionality of each of them.

3.1 Cache Attacks

Cache attacks take advantage of cache collisions occurring in some shared cache in the cache hierarchy. These collisions are detected by measuring access times, i.e., by distinguishing accesses between two levels in the same cache hierarchy or between accesses to the cache hierarchy and accesses to the DRAM. Although several attack designs have been proposed, two (and their combinations) stand out over the rest: the `Flush and Reload` and the `Prime and Probe` attacks.

```
1  int probe(char *adrs) {
2    volatile unsigned long time;
3
4    asm __volatile__ (
5      "  mfence             \n"
6      "  lfence             \n"
7      "  rdtsc              \n"
8      "  lfence             \n"
9      "  movl %%eax, %%esi  \n"
10     "  movl (%1), %%eax   \n"
11     "  lfence             \n"
12     "  rdtsc              \n"
13     "  subl %%esi, %%eax  \n"
14     "  clflush 0(%1)      \n"
15     : "=a" (time)
16     : "c" (adrs)
17     :  "%esi", "%edx");
18   return time < threshold;
19 }
```

Fig. 1. Flush and Reload code snippet from [40]

Flush and Reload: The `Flush and Reload` attack assumes memory sharing between victim and attacker and is performed in three major steps. In the first

step, the attacker removes a shared memory block from the cache hierarchy (e.g., with the `clflush` instruction. In the second step, the attacker lets the victim interact with the shared memory block by waiting a specified number of cycles. In the last step, the attacker re-accesses the removed memory block. If the memory block comes from the cache (i.e. a fast access time observed) she derives that the victim utilized the memory block during the waiting period. On the contrary, if the memory block comes from the DRAM (i.e. a slow access observed) the attacker assumes the victim did not access the memory block during the waiting period. Flush and Reload attacks have been exploited in [40, 20, 12]

Prime and Probe: The `Prime and Probe` attack does not make any special assumption between victim and attacker (rather than sharing the underlying hardware) and is also performed in three major steps. In the first step, the attacker fills a portion of the cache (usually a set) with his own data. Then the attacker waits again hoping to observe activity from the victim. In the third step, he re-accesses the data he used to fill part of the cache. If all his data comes from the cache (i.e. low access times) the attacker assumes the victim did not use the portion of the cache she filled (otherwise one of her data lines would have been evicted). On the contrary, if she observes that at least one of her data lines (i.e. high access times) comes from the DRAM, the she assumes the victims utilized that portion of the cache. Prime and Probe attacks were exploited in [18, 9]

Combinations of both techniques have also shown to be successful at executing attacks. For instance, a Prime and Reload approach would be successful in those systems in which users do not have access to a `clflush` like instruction.

3.2 DRAM Access Attacks

The DRAM is usually divided in channels (physical links between DRAM and memory controller), Dual Inline Memory Modules (DIMMS, physical memory modules attached to each channel), ranks (bank and front of DIMMS), banks (analogous to cache sets) and rows (analogous to cache lines). If two addresses are physical adjacent they share the same channel, DIMM, rank and bank. Additionally each bank contains a row buffer array that holds the most recently accessed row.

In fact DRAM access side channel attacks take advantage of collisions produced between addresses physically adjacent, i.e., in the same bank, rank, DIMM and channel. More precisely, retrieving a memory location from the row buffer yields faster accesses than retrieving it from the bank. In fact, the row buffer acts like a direct mapped cache holding the most recently accessed rows. In order to build a successful attack, an attacker would first have to prime the row buffer. Then he would have to evict the cache portion that the target memory location occupies, making sure that the next victim access will hit the DRAM. After the victim access, the attacker probes the row buffer to check whether the victim accessed memory locations within the same bank. If he did, he will observe row buffer misses, and thus an increase in the access time. On the contrary, if the victim did not access bank congruent locations, the attacker will obtain fast accesses for his primed data. The attack was proposed and exploited in [29].

3.3 Rowhammer Attacks

Rowhammer attacks take advantage of disturbance errors that occur in adjacent DRAM rows within the same DRAM bank. It has been demonstrated that continuous accesses to a DRAM row can indeed influence the charge of adjacent row cells, making them leak at a higher rate. If these cells loose charge faster the charge refreshing interval, accesses to a DRAM row induce bit flips in adjacent rows in the same bank. Note that bit flips can have catastrophic consequences, e.g., cryptographic fault injections.

In order to execute a rowhammer attack, an attacker performs three essential steps. The attacker first opens a row that resides in the same bank as the row in which bit flips want to be induced. Then the attacker performs accesses to the DRAM row, trying to influence the charge leak rate of adjacent rows. In the third step, the attacker removes the accessed DRAM row from the cache hierarchy to ensure the next subsequent accesses will access the DRAM (e.g. with the clflush instruction). Note that steps 2 and three are continuously executed in a loop to try to induce bit flips in the victims DRAM row. Examples of studies in rowhammer attacks are [23, 11, 7].

```
code1:
  mov (X), %eax
  mov (Y), %ebx
  clflush (X)
  clflush (Y)
  mfence
  jmp code1
```

Fig. 2. Rowhammer code snippet from [23]

3.4 Memory Bus Locking Covert Channels

Memory Bus Locking attacks take advantage of pipeline flushes that occur when atomic operations are performed on data that occupies more than one cache line. However, if the atomic operation is performed on data that fits within a single cache line the system locks the cache line for the atomic operation to happen. The pipeline flushing operations cause pipeline flushes that incur in performance overheads. Memory bus locking mechanisms have been used as a method to establish covert channels and derive co-residency in IaaS clouds and as a mechanism to perform Quality of Service degradation (QoS) [35, 17].

4 Implicit Characteristics of Microarchitectural Attacks

In this section we review those characteristics that well known microarchitectural attacks exhibit in their design. More in particular, we put our focus in three of the most dangerous microarchitectural threats presented in the last years: Cache attacks, memory bus locking and rowhammer attacks.

4.1 Cache Attacks

As already explained in the background section, cache attacks are implemented by creating cache contentions in any of the caches in the cache hierarchy. This effect, depending on the attack structure, can be accomplished in several ways. However, all cache attacks have three main characteristics in common:

- **High resolution timers** Cache attacks rely on the ability of distinguishing different access patterns (e.g. L1 accesses from LLC accesses or even memory accesses). As these accesses differ at most by a few hundred cycles, the attacker needs to have access to a timer accurate enough to carry out an attack. These timers can be accessed in many different ways (e.g., the common `rdtsc` function or an incremental thread).
- **Memory Barriers** Another common factor of cache attacks is the utilization of memory barriers to ensure serialization before the targeted reads are executed (i.e. making sure any load and stores have finished before the target access). This memory barriers sometimes can sometimes be embedded in timer instructions (like the popular `rdtscp` instruction) or can come in the form of different instructions.
- **Cache evictions** The last factor that all cache attacks have in common is the implementation of an eviction routine that removes a target cache line from the cache hierarchy. This can be implemented with the popular flush instruction (in the case of shared memory) or with the creation of eviction sets.

4.2 DRAM Access Attacks

Similar to cache attacks, DRAM access attacks base their functionality on the capacity of colliding with an attacker in the same row buffer of the same bank, rank, Dual Inline Memory Module (DIMM) and channel. In order to achieve that purpose, DRAM attacks share common characteristics with cache attacks.

- **Cache evictions** DRAM access attacks exploit collisions in the DRAM row buffer, and as such, attackers need the victim to access the DRAM (instead of caches) for memory fetches. Since every time the victim makes an access the memory location would be brought to the cache, the attacker continuously needs to evict memory location from the cache hierarchy. As with cache attacks, this can be achieved by utilizing specific instructions (i.e. the flush instruction) or with carefully designed eviction set mechanisms.
- **Fine grain timers** In order to retrieve meaningful information, attackers need to distinguish between DRAM row buffer hits and misses. This implicitly requires accesses to fine grain timers.
- **Memory barriers** To prevent out of order execution and obtain precise timing behavior.

4.3 Rowhammer Attacks

Perhaps the microarchitectural attack with the most dangerous implications that has been discovered in the last decade, due to the ability of flipping bits from memory locations belonging to a victim. In contrast to cache or memory bus locking attacks, rowhammer attacks only have one clear distinguishable characteristic distinguishable from bening code:

- **Cache evictions** In fact, rowhammer attacks rely on continuous access to a DRAM location that shares the DRAM bank with the victims memory location. Thus, attacker need to continuously bypass the cache, otherwise the CPU will bring the accessed DRAM portion to the cache for faster access. There are several methodologies to avoid the cache accesses, some of which are similar to those utilized in cache attacks (e.g. flush instructions or eviction sets).

4.4 Memory Bus Locking Covert Channels

Another popular attack in the last years is the ability to stall the memory bus to establish a covert channel between two co-resident processes. As with cache attacks, memory bus locking covert channels also have their own characteristics that need to be accomplished in their design:

- **High resolution timers** As with cache attacks, memory bus locking mechanisms require of a fine grain timer that measure the transmission of a 1 or 0 bit depending on the time to execute atomic operations. Further, fine grain timers might also be utilized by attackers willing to ensure the effects of the memory bus lockdown prior to its execution.
- **Lock Instructions** In addition to the fine grain timer, these attacks also require of specific atomic instructions capable of locking the memory bus. In x86 systems this instructions include, among others, the ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, DEC, FADDL, INC, NEG, NOT, OR, SBB, SUB, XADD, XOR instructions with the lock prefix. Further, the XCHG instruction executes atomically when operating on a memory location, regardless of the LOCK use.

5 Our Approach: MASCAT a Static Analysis Tool for Microarchitectural Attacks

Once the main characteristics of the attacks we are trying to prevent have been identified, we propose a novel technique based on statically analyzing binaries looking for embedded microarchitectural attacks. Proposed countermeasures to detect microarchitectural attacks are based on dynamically detecting specific microarchitectural patterns when the binary is executed. We observe two clear problems with this mechanism:

– Cache attacks can be embedded into a binary to be executed only after some specific time/date. Thus, the system should be monitoring the microarchitectural patterns of the binary every time it is executed and not only once. This can create significant overhead in the system.
– The knowledge on how to install such a mechanism might not be trivial for every end-user. Thus, although such an approach might well be adoptable by large scale systems (like IaaS clouds), a regular user might not have the capabilities to perform such an action.

Our approach tries to solve both issues. Imagine a regular user that receives a binary from a trustworthy (or not) person, or an official application distributor aiming at providing malware free applications. How do they know whether the binary contains malicious code, and particularly microarchitectural exploitation code? The most common approach is to utilize an antivirus software to determine whether the binary is safe to execute. Although antivirus tools might work well with certain kind of malware [33], their success when detecting microarchitectural attacks is still an open question. Table 1 shows the outcome of such an analysis in list of well known antivirus software (the best in 2016, as stated in [24]). Indeed, we utilized several examples of both cache, DRAM, rowhammer and memory bus locking attacks. None of the antivirus softwares was able to detect that the binaries were malicious.

Antivirus software	Output cache/DRAM attack	Output rowhammer	Output bus locking
Avast	✓	✓	✓
BitDefender	✓	✓	✓
Emsisoft	✓	✓	✓
ESET-NOD32	✓	✓	✓
KasperSky	✓	✓	✓
F-secure	✓	✓	✓
McAfee	✓	✓	✓
Symantec	✓	✓	✓
TrendMicro	✓	✓	✓

Table 1. As of 12.28.2016, none of the major antivirus tools detects any of the attacks we analyzed.

In order to cope with this problem, we propose MASCAT, a tool that tries to find intrinsic characteristics of microarchitectural attack code in potentially malicious binaries. MASCAT consists on several fully automated scripts that are executed by the well known IDA Pro static binary analyzer. Thus, MASCAT can be directly adopted by any person using IDA Pro or can be directly translated by any other static binary analyzer. Furthermore, it can be offered as an online scanner for microarchitectural attack code. The following section summarize the approaches that our static analysis approach follows to detect malicious microarchitectural attacks.

5.1 Attributes Analyzed by MASCAT

This section summarizes the attributes that MASCAT tries to identify within the binary code. Note that the goal of our design is not to detect all possible microarchitectural attack designs (indeed, if an expert attacker knows the approach taken by our tool, he can always find a way to bypass it) but rather to give a good framework on the detection of existing attacks. More attributes can be added to cover more sophisticated and intelligent designs of microarchitectural attacks.

- **clflush instruction:** The `clflush` instructions is a common factor to both cache attacks, DRAM access and rowhammer attacks. Since these attacks require several accesses to succeed, our tool identifies those `clflush` instructions that are being used inside a loop.
- **monvnti & movntdq instructions:** The `monvnti` and `movntdq` allow attacker to bypass the cache to directly access the DRAM in rowhammer attacks. Our tool further tries to find these instructions being utilized in a loop.
- **Thread counters, performance counters, rdtscp & rdtsc instructions:** The `rdtscp` and `rdtsc` instructions are fine grain timers provided in x86_64 processors. In contrast to `rdtsc`, `rdtscp` further implements memory barrier instructions before reading the clock cycle counter. Similarly, the performance counters can give us the number of clock cycles elapsed/evictions. Lastly, if none of these is accesible, a timer can be created by using a thread continuously incrementing a counter. Our tool identifies all these approaches being utilized in loops.
- **lfence & mfence & cpuid instructions:** Instructions that execute memory barrier operations to ensure all loads (lfence) and all loads and stores (mfence, cpuid) have been issued. This instructions are necessary to prevent out of order execution and to obtain accurate timings from the fine grain timers.
- **lock instructions:** Atomic instructions that can be issued to implement memory bus locking attacks. Our tool monitors for all the instruction with the lock prefix plus the XCHG instruction.
- **jump opcodes:** L1 instruction cache attacks are commonly designed utilizing jump opcodes to jump to set concurrent addresses. Our tool again analyzes whether any of the functions inside a binary assigns jump opcodes in a loop.
- **Pointer chasing & Page size jump approaches:** One of the approaches that can be taken to create cache eviction sets (which can be utilized for cache, DRAM access and rowhammer attacks) is the pointer chasing approach, in which the address of the next address is stored in contents of the previous address. Another approach is to introduce jumps of łnumber of bytes within a vector, being łthe necessary bytes to find set-concurrent addresses. Our tool tries to identify both approaches being utilized in binary code.

- **selfmap translation & slice mappings** Although non-available from userspace since linux kernel 4.0.0, prior kernels are still vulnerable from having an attacker looking at the phsyical address of his memory to facilitate eviction set creations. Aiming at avoiding these attacks, our tool also identifies accesses to this particular mapping. Further, our tool also finds whether the *known* slice selection algorithms are utilized in the binary code to guess the slice location of the memory addresses.
- **affinity assignment** Some of the above mentioned attacks (like the L1 cache attacks or LLC attacks) only succeed when core co-residency or CPU cluster co-residency are achieved. Further, the timers (and specially the thread counter) exhibit better performance when scheduled in a single core. Thus, our tool also tries to identify whether there is any affinity assignment inside the inspected binary code.

5.2 Measuring the Threat Score

One of the most important challenges of the design of our tool is to determine, based on the attributes observed, whether a threat exists or not. Although we could use machine learning algorithms to design such a methodology, we believe there are several facts that have to be smartly taken into account (e.g. location of the attributes, location of the nested functions calling the functions where the attributes were found, etc.) that can make machine learning algorithms to perform poorly. In contrast, we design MASCAT to be intelligent enough to retrieve those facts, such that we do not have a necessity for utilizing an automated data analysis tool. We decide to implement a score system based on the combination of attributes observed. For instance, we know memory barrier instructions are not a threat if they are not issued together with timers, lock or eviction instructions. In this sense, we designed the following score based threat classification:

Maximum Threat Attributes: Attributes from this category are immediately considered a threat when found by our algorithm. `clflush, monvnti, movntdq` and selfmap translation attributes fall into this category. The first 3 because they can directly imply a rowhammer attack, the latter one because having access to the physical address space can become a threat for both cache, DRAM access rowhammer attacks. The location where these attributes (and the functions calling them) are found will be marked in red.

Medium Threat Attributes: Attributes from this category are not immediately considered threats but rather warnings. The locations in which these attributes are found will only be considered a threat if the rest of the necessary attributes (as revealed in section X) are found within the same location of the code. Examples of these attributes are `rdtscp` and `lock` instructions, as well as pointer chasing, set-concurrent jumps and jump opcode assigments. A combination of 2 related medium threat attributes will immediately be considered a threat, while a combination of two non-related medium threat attributes will

not change the score. For instance, `rdtscp` and `lock` instructions together would be considered a threat, while `lock` instructions with pointer chasing functions are not (see Section).

Low Threat Attributes: Attributes from this category are not immediately considered threats but rather small warnings. Only if these attributes are observed within the same location of the necessary of necessary medium or low threat attributes the code portion is considered a threat. Examples of these kind of attributes are memory barriers, timers (excluding `rdtscp`) and affinity assignment. Only if two low threat attributes are observed with a medium threat attribute the code location is classified as a threat.

5.3 Tool Framework

In order to implement the specified design we utilize the popular IDA PRO static binary analyzer. We chose IDA because it offers us a high level programming language and several built in functions that facilitate our attribute finding design. Note that developing our tool with another open source static library analyzer (like hooper) should also be feasible with an extra amount of work.

In summary MASCAT works by analyzing binary elf files *without the need for the source code*. The tool tries to find the attributes described above, coloring those locations where microarchitectural attack related features are found. Our tool colors threats in read, warnings in orange and small warnings in magenta. Note that, as MASCAT finds new attributes in already colored locations, the colors of those locations might change. Once the analysis is finished, MASCAT outputs a summary of the locations in which threats and warnings were found and their root of cause (e.g., `rdtsc` instruction, `lock` instructions, etc.). Figure 3 shows a visual example of an analysis made to one of our microarchitectural attacks. The binary is considered a threat for utilizing flush and fine grain timer instructions.

6 Experiment Setup

The goal of our experimental setup is to test different sets of microarchitectural attacks designed by very different researchers. Thus, our analysis includes the Mastik tool (which performs all range of L1 and LLC attacks) designed by Yarom et al. [39], LLC attacks and memory bus locking attacks designed by Inci et al. [14] and Irazoqui et al. [20, 18, 16], LLC, cache prefetching, DRAM access and rowhammer attacks designed by Gruss et al. [12, 29, 10, 11], rowhammer attacks designer Google Project Zero (which were modified to cover [31]), L1 cache attacks from Tampere university [1] and DRAM access attacks designed by Abraham Fernandez, former Intel employee. Some of these source codes were modified to cover several corner cases (e.g., thread created timers, eviction set creations, non-temporal accesses, doble-sided rowhammer, etc.). In total, we cover a range of 26 different microarchitectural attacks.

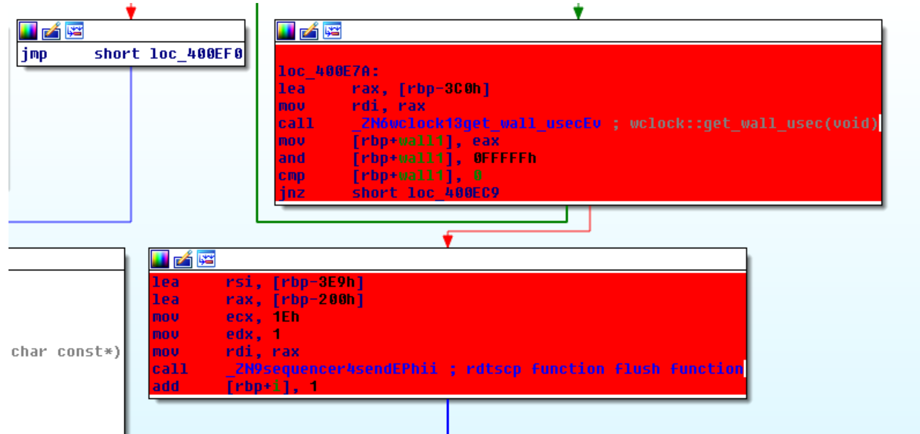

Fig. 3. Visual example output of MASCAT

As the benign binary set that was included for our test framework, we chose 100 binaries from the phoronix-test-suite. This choice was not random, but was intended to possibly maximize the number of false positives that our tool can output. In fact, a large number of the tests in the phoronix-test-suite are performance benchmarks that might need to utilize some of the attributes that our tool looks for. Since we do not expect to observe such a behavior in regular non-performance measurement related binaries, the binaries from the phoronix-test-suite should give a good upper false positive threshold.

Our analysis only focused in x86_64 binaries and utilized binaries compiled under Linux Operating Systems (as they are the most utilized OS to carry out microarchitectural attacks). However, this does not mean that other binaries (e.g., smartphone applications) can not be similarly analyzed. In order to perform such an analysis, only the semantics related to each architecture/OS should change (e.g., in ARM there is not a `clflush` instruction but the `clean_cache` system call). For that reason, we believe the same analysis can be easily carried out for other architectures and scenarios.

7 Results

The results obtained for both the malicious and benign code in terms of false positives and negatives indeed depends on the conditions (i.e., strength factor) imposed for the attributes found. We start by taking the approach described in Section, i.e., imposing the conditions to detect all the microarchitectural attacks described in Section. The effects on the success rate that different conditions have will discussed in the subsequent paragraphs.

Our analysis indicates that all the malicious binaries that we analyzed flag as threats. The reason for this categorization is indeed very different in each of the binaries (e.g. rowhammer codes tend to flag due to the utilization of specialized

eviction instructions, while cache attacks are usually a combination of timers and eviction policies). In any case, we obtained a 100% success rate in the 25 malicious binaries analyzed. Note that our tool looks for the attributes in a generic manner, i.e., our analysis scripts are not modified accordingly to flag each malicious binary as a threat.

As for the benign binaries, the results are presented in Table 7, in which binaries categorized as threats are displayed with red font, binaries with warnings are displayed with yellow font and non-warning binaries are displayed with green font. We observed that out of the 100 binaries, only 6 flagged as potential threats. The rest either contained insufficient attributes to be considered as threats (i.e. warnings) or did not contain any of the attributes we were looking for. Thus, we obtained a 6% of false positives and a 0% of false negatives in our test binaries. Note that, due to the nature of the phoronix-test-suite benchmarks, the 6% rate of false positives is a good upper threshold indicator, but we expect a lower false positive rate with regular non-benchmarking binaries.

Table 7 shows the reason why those 6 binaries flagged as threats. Nqueens, multichase, mencoder and fs_mark benchmarks flagged due to the usage of non-temporal load and stores instructions that could be an indicator of rowhammer attacks. Fio and ffmpeg benchmarks flagged because of the usage of timers, memory barriers and pointer chasing/set congruent jump instructions within the same location, which could be indicators of cache and DRAM access attacks. Lastly fs_mark and again ffmpeg flagged because of the usage of timers and atomic instructions, indicating a potential memory bus locking covert channel exploitation.

Obviously this means that our false positive rate and our false negative rate will increase/decrease depending on the importance assigned to the attributes found. Table 7 represents the False Negatives (FN) and False Positives (FP) when the score of the attributes are changed to look only for a subset of attacks from our attack vector (rowhammer, memory bus locking, DRAM/cache attacks). We cover DRAM access and cache attacks together, as they share exploitation attributes. Our results show that, due to common characteristics between cache/DRAM access and rowhammer attacks, looking only for rowhammer also gives us zero false negatives for cache/DRAM attack binaries. Finally, if only cache/DRAM access attributes are seeked, as cache evictions would not directly be considered a threat without the appropriate timers, only 50% of the rowhammer attacks are detected. On the contrary, the nature of bus locking covert channels makes our tool miss cache/DRAM and rowhammer attacks if we only look for memory bus locking characteristics. Finally, we further observe that the false positive rate drops if we only look for one of the attacks, being even as low as 2% in the case of cache attacks. In summary, the attributes searched for and their importance critically influence the false negative and false positive rate found in malicious and benign code respectively. In any case, covering the 4 microarchitectural attack range only lead us to 6% false positive rate in benign benchmarking code. We believe the false positive rate should be even smaller if benign code with other characteristics was chosen. However our choice was

aio-stress	apache	api-test	apitrace	blake
blogbench	botan	byte	c-ray	cairo
clomp	compress-7zip	compress-lzma	pbzip	core-breach
crafty	cyclictest	d-bench	dcraw	dolfyn
doom	ebizzy	encode-ape	encode-ape	encode-flac
encode-mp3	ffmpeg	ffte	fftw	fhourstones
fio	fs_mark	glmark2	gluxmark	gmpbench
gputest	graphics_magic	gtkperf	himeno	hint
hmmer	hpcc	hpcg	interbench	iozone
johnripper	juliagpu	jxrendmart	lamps	lightsmark
luxmark	mandelgpu	mafft	mencoder	minion
mrbayes	multichase	nbp	nginx	nqueens
openarena	openporous	padman	parboil	pgpbench
php	polybench	postmark	prey	primesieve
psstop	qdvpau	ramspeed	reaction	redis
render-bench	rodinia	sample_program	scimark2	shoc
smallpt	smallptGPU	somkingguns	specviewperf	sqlite
stockfish	stream	stresscpu	supertuxart	systester
tachyon	tesseract	tremulous	tscp	ttsiod_renderer
unigine_valley	unigine_heaven	unigine_sanctuary	unigine_tropics	viennacl

Table 2. Output of the microarchitectural static analyzer on 100 benign phoronix-test-suite binaries. Green indicates no warnings, yellow indicates the presence of warnings and red indicates presence of threats

Binary flagged	Reason
ffmpeg	timers, memory barriers, pointer chasing and lock functions
fio	timers, memory barriers and pointer chasing functions
fs_mark	timers, lock and non-temporal load/store functions
mencoder	non-temporal load/store functions
multichase	non-temporal load/store functions
nqueens	non-temporal load/store functions

Table 3. Explanation for benign binaries classified as threats

Attack targeted	FN rowhammer	FN cache/DRAM	FN bus locking	FP bening
rowhammer	0%	100%	0%	4%
cache/DRAM attacks	50%	100%	0%	2%
bus locking	100%	0%	100%	2%

Table 4. False Negative/Positive variation looking only for a subset of microarchitectural attacks

intended to obtain an estimate of the upper false positive threshold that our tool can output.

8 Conclusion

For the first time we presented a static code analysis tool, MASCAT capable of scanning for common microarchitectural attacks. Microarchitechtural attacks are particularly damaging as they are hard to detect since they use standard resources, e.g. memory access, made available to applications at the lowest privilege level. Given the rise of malicious code in app stores and online repositories it becomes essential to scan applications for such stealthy attacks. The proposed tool, MASCAT is ideally suited to fill this need. MASCAT can be used by app store service providers to perform large scale fully automated analysis of applications. The initial MASCAT suite is built to include attack vectors to cover popular cache/DRAM access attacks, rowhammer and memory bus covert channels. Nevertheless, our tool is easily configurable to include newer attack vectors as they arise.

References

1. Aciiçmez, O., Brumley, B. B., and Grabher, P. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings* (2010), S. Mangard and F.-X. Standaert, Eds., vol. 6225 of *Lecture Notes in Computer Science*, Springer, pp. 110–124.
2. Aciiçmez, O. Yet Another MicroArchitectural Attack: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*.
3. Aciiçmez, O., K. Koç, c., and Seifert, J.-P. Predicting secret keys via branch prediction. In *Topics in Cryptology CT-RSA 2007*, vol. 4377. pp. 225–242.
4. Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association, pp. 53–70.
5. 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 *CHES* (2014), pp. 75–92.
6. Bernstein, D. J. Cache-timing attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.
7. Bhattacharya, S., and Mukhopadhyay, D. *Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis*.
8. Brasser, F. F., Davi, L., Gens, D., Liebchen, C., and Sadeghi, A. Can't touch this: Practical and generic software-only defenses against rowhammer attacks. *CoRR abs/1611.08396* (2016).
9. Fangfei Liu and Yuval Yarom and Qian Ge and Gernot Heiser and Ruby B. Lee. Last level cache side channel attacks are practical. In *S&P 2015*.
10. Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 368–379.

11. Gruss, D., Maurice, C., and Mangard, S. Rowhammer.js: A remote software-induced fault attack in javascript. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721* (New York, NY, USA, 2016), DIMVA 2016, Springer-Verlag New York, Inc., pp. 300–321.
12. Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium* (2015), USENIX Association, pp. 897–912.
13. Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. SP '11, pp. 490–505.
14. İncİ, M. S., Gülmezoglu, B., Eisenbarth, T., and Sunar, B. Co-location detection on the cloud. In *COSADE* (2016).
15. İnci, M. S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., and Sunar, B. *Cache Attacks Enable Bulk Key Recovery on the Cloud*. 2016.
16. İncİ, M. S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., and Sunar, B. Cache attacks enable bulk key recovery on the cloud. In *Cryptographic Hardware and Embedded Systems–CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings* (2016), vol. 9813, Springer, p. 368.
17. Inci, M. S., Irazoqui, G., Eisenbarth, T., and Sunar, B. Efficient adversarial network discovery using logical channels on microsoft azure. In *Annual Computer Security Applications Conference* (Los Angeles, CA, US, dec 2016).
18. 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 *36th IEEE Symposium on Security and Privacy (S&P 2015)*.
19. Irazoqui, G., Eisenbarth, T., and Sunar, B. Cross processor cache attacks. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security* (2016), ASIA CCS '16, ACM.
20. Irazoqui, G., İncİ, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID* (2014), pp. 299–319.
21. Irazoqui, G., İncİ, M. S., Eisenbarth, T., and Sunar, B. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (2015), ASIA CCS '15, pp. 85–96.
22. Kim, T., Peinado, M., and Mainar-Ruiz, G. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 189–204.
23. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 361–372.
24. Best 2016 antivirus.
25. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* (2016), pp. 549–564.
26. Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., and Lee, R. B. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE Symposium on High-Performance Computer Architecture* (Barcelona, Spain, mar 2016), pp. 406–418.

27. Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1406–1418.
28. Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06.
29. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. Drama: Exploiting dram addressing for cross-cpu attacks. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association, pp. 565–581.
30. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. DRAMA: exploiting DRAM addressing for cross-cpu attacks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* (2016), pp. 565–581.
31. Qiao, R., and Seaborn, M. A new approach for rowhammer attacks. *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) 00*, undefined (2016), 161–166.
32. 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 *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pp. 199–212.
33. Sukwong, O., Kim, H. S., and Hoe, J. C. Commercial antivirus software effectiveness: an empirical study. *Computer 44*, 3 (2011), 0063–70.
34. van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., and Giuffrida, C. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1675–1689.
35. Varadarajan, V., Zhang, Y., Ristenpart, T., and Swift, M. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 913–928.
36. Wang, Z., and Lee, R. B. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (2007).
37. Xiao, Y., Zhang, X., Zhang, Y., and Teodorescu, R. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association, pp. 19–35.
38. Xu, Z., Wang, H., and Wu, Z. A measurement study on co-residence threat inside the cloud. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 929–944.
39. Yarom, Y. Mastik: A micro-architectural side-channel toolkit, 2016.
40. Yarom, Y., and Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 719–732.
41. Zankl, A., Miller, K., Heyszl, J., and Sigl, G. Towards efficient evaluation of a time-driven cache attack on modern processors. In *ESORICS 2016*.
42. Zhang, T., Zhang, Y., and Lee, R. B. *CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds.* 2016.

43. Zhang, T., Zhang, Y., and Lee, R. B. Memory dos attacks in multi-tenant clouds: Severity and mitigation. *CoRR abs/1603.03404* (2016).
44. Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.
45. Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*.
46. Zhou, Z., Reiter, M. K., and Zhang, Y. A software approach to defeating side channels in last-level caches. In *CCS 2016*.