HexPADS: a platform to detect “stealth” attacks

Mathias Payer

Purdue University

Abstract. Current systems are under constant attack from many different sources. Both local and remote attackers try to escalate their privileges to exfiltrate data or to gain arbitrary code execution. While inline defense mechanisms like DEP, ASLR, or stack canaries are important, they have a local, program centric view and miss some attacks. Intrusion Detection Systems (IDS) use runtime monitors to measure current state and behavior of the system to detect an attack orthogonal to active defenses.

Attacks change the execution behavior of a system. Our attack detection system HexPADS detects attacks through divergences from normal behavior using attack signatures. HexPADS collects information from the operating system on runtime performance metrics with measurements from hardware performance counters for individual processes. Cache behavior is a strong indicator of ongoing attacks like rowhammer, side channels, covert channels, or CAIN attacks. Collecting performance metrics across all running processes allows the correlation and detection of these attacks. In addition, HexPADS can mitigate the attacks or significantly reduce their effectiveness with negligible overhead to benign processes.

1 Introduction

Software is constantly under attack using a wide set of attack vectors. The attack surface increases as more devices go online. Connected devices expose running services but also request services from untrusted parties through potentially vulnerable client-side software like web browsers.

Current systems leverage a wide range of different attack detection and protection mechanisms, many of them in combination. Protection mechanisms like Address Space Layout Randomization (ASLR) [21], Data Execution Prevention (DEP) [27], stack canaries [12] protect against some memory corruption attacks. Host-based protection mechanisms mitigate exploitation attempts of unknown or unpatched vulnerabilities in software but terminate the application whenever an attack is detected. Patching removes the vulnerability and mitigates attacks. Unfortunately, patches are not readily available when a vulnerability is disclosed. Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) on the other hand detect an attack before, during, or after it happened. Commonly, intrusion detection systems measure a set of parameters and check if the fingerprint matches any of the known signatures (note that signatures can be Turing complete verifiers). Network-based IDS like Bro [22] match network packets against known signatures and alert if an attack is detected. Host-based IDS collect information about a system and match this information against a set of rules or attack signatures. An IDS is either misuse-based, matching observed behavior with a set of attack signatures or anomaly-based, detecting divergences.

Existing host based defense mechanisms focus on memory corruption and code reuse attacks but offer limited to no protection against information leaks, side channel, and covert channel attacks. Existing host-based IDS detect a set of individual attacks by matching fingerprints of individual attacks against the runtime collected statistics but are limited to the collected software metrics. Due to the limited software metrics provided by the operating system itself, memory-based attacks like side channels and covert channels cannot be observed directly and are therefore *stealthy* (to available metrics). Software under attack it behaves differently compared to a regular execution. Lightweight, low performance overhead program analysis tools like performance counters (both hardware-based and software-based) allow a detailed fingerprinting of the execution behavior of software. We leverage the information collected from a set of specific probes to detect attacks through their anomalies, matching execution behavior of processes against attack classes. Using additional runtime metrics from the performance counters allows us to uncover these otherwise undetected attacks.

We propose HexPADS, a host-based, Performance-counter-based Attack Detection System that measures performance characteristics of all processes and detects attacks by matching a set of signatures. HexPADS is especially apt at detecting long running Covert and Side Channel (CSC) attacks. Compared to per-attack signatures, HexPADS uses broader per attack-vector signatures, generalizing signatures to all attacks in an attack class whenever possible, e.g., protecting against all CSCs by detecting cache performance anomalies instead of detecting specific cache attacks. HexPADS collects statistics about running processes and measures common performance parameters using existing low-overhead, hardware-based performance counters. To our knowledge, HexPADS is the first IDS that leverages per-process performance counters to detect attacks. In our evaluation we show that our prototype implementation achieves negligible (non-measurable) overhead and in a set of case studies we show how HexPADS detects (and mitigates) rowhammer [25], CSCs [24,39,37,18,11], and CAIN [2] attacks. Side-channel based information leaks are used to extract data from running systems and processes or to corrupt memory in the case of rowhammer. Such memory CSC attacks can, e.g., be used to break AES cryptographic key generation, or to break ASLR in the cloud [2]. The main contributions are:

1. Design of HexPADS, a host-based attack detection system that detects stealth attacks through fine-grained process monitoring using performance counters and performance metrics exported by the kernel.
2. Evaluation of a prototype implementation of our attack detection system that detects cache attacks, DRAM attacks like rowhammer, and memory deduplication attacks like CAIN at negligible overhead.
3. A discussion of mitigation mechanisms that protect against cache, DRAM, and memory deduplication attacks.

2 Threat model and attacker goals

We assume a powerful threat model where the attacker can execute user-level code on the system. An attacker can achieve these capabilities either through a legitimate service on the system that offers the computational capabilities or through the exploitation of a service. HexPADS configures performance counters for all processes. To ensure integrity of our monitor, we assume that HexPADS is running as a separate process at higher privileges than the attacker and that the attacker cannot access the monitor or disable performance monitoring.

The trusted computing base contains the underlying hardware, hypervisor, and operating system. An alternative, hypervisor-based implementation would remove the operating system from the trusted computing base. We assume that the attacker does not have raw memory access and that we can rely on the performance counter results. We trust the integrity of memory, assuming that we detect attacks like rowhammer *before* memory is corrupted.

The attacker's goals are to escalate privileges, to communicate with other processes, to leak information, or to execute code while remaining undetected. HexPADS continuously monitors the system and detects an ongoing attack. Attack detection is inherently restricted to the precision of the measured runtime characteristics and limited by the effectiveness of the monitor to distinguish between benign behavior and attacks.

3 Background

HexPADS leverages existing process metrics and performance counters to collect information about all running processes. Both process metrics and performance counters are available and supported on all major operating systems. Here we give, without loss of generality, an overview of process metrics and performance counters on Linux systems.

3.1 Process metrics

Operating systems continuously collect basic information about all running processes. This information is exposed to user-space to administer processes and to diagnose problems with user-space utilities. Linux provides the `/proc` pseudo-filesystem as an interface to kernel data structures which are accessible from user-space. The files in the exported directory are mostly read-only and used for informative purposes but kernel settings can be changed by writing to these files as well. Most Linux distributions make the `/proc` directory accessible to user-space processes, exposing information about all running processes.

Each running process has its own directory under the root `/proc` directory named after the process' PID. The file `stat` contains a wide range of process metrics, including name of the executable, process state, the PID of the parent, the process group, the associated terminal, the amount of page faults, total execution time in both user and kernel space, priority, number of threads for this process, when the process was started, memory limits for regions like heap or stack, which processor the task runs on, and the scheduling policy[1]. HexPADS collects all `stat` information.

The recommended way of using this information is to scrape all numerical directories in the `/proc` directory, thereby iterating over all running threads and processes. Tools like `ps`, `top`, or `killall` all leverage the files in the `/proc` directory to fulfill their tasks.

3.2 Performance counters

Hardware performance counters are available in all major CPU architectures. These performance counters are special-purpose registers that collect information about the executed instructions. The names of the counted events differ between platforms and the number of available registers (and thereby the amount of performance events that can be sampled at the same time) is platform specific with low-end architectures generally featuring less performance counting infrastructure. An advantage of using hardware performance counters is that the overhead to count specific events is negligible (as the hardware is responsible for all the heavy lifting). The individual counters and their configuration are managed by the Performance Monitoring Unit (PMU).

The Intel x86 platform offers detailed configurable performance counters since the Intel Pentium. The Intel Core i7 family supports base level and enhanced architectural performance monitoring with four general-purpose, configurable performance counters (i.e., four types of events can be counted on any core at any point in time) [4, chapter 18.2]. In addition to counting, the Intel architecture also supports precise event-based sampling. Instead of counting the occurrences of an event, the PMU also takes a snapshot of the processors state at the time of the event. On x86, such a snapshot consists of the instruction pointer, stack pointer, and all general purpose registers. AMD processors have similar counters and hardware capabilities.

On Linux, the PMU can be configured using the `perf_event_open` system call (which does not have a libc-based wrapper but needs to be called using inline assembly). Some user-space programs, e.g., `perf` provide a command-line interface to the PMU and allow the collection of detailed performance events for executing software. The Linux `perf_event` interface tries to unify performance counter access across architectures and processor families. Performance counters can be assigned system-wide or per-process with a wide range of conditions (e.g., the processor the task runs on). After setting up the PMU, the event can be configured using the `ioctl` system call. Samples can be read explicitly by polling through a `read` system call or implicitly by setting up a signal that is delivered whenever the counter reaches a pre-defined value (or the buffer used to store the samples when sampling overflows).

The only additional overhead when using hardware-based performance counters comes from (i) configuring the PMU whenever a process is scheduled and (ii)

[1] Additional information and details are available on the `proc` manpage.

updating the aggregates whenever the process is interrupted. Collecting counters might incur some overhead during execution but these effects are hidden by the microarchitecture. In addition, if an event is sampled (and not just counted) then there is also additional cache pressure when samples are written into the sample buffer. The overhead of running performance counters alongside the executed software is in the noise (less than 1%).

4 HexPADS design

The core principle of HexPADS is to search for general attack behavior and attack artifacts in all running processes. The underlying hypothesis is that software attacks significantly change the environment or the behavior of a process or processes. Both the attacking process (if run on the same machine) and the attacked process (usually a service) will exhibit behavior that can be mapped to an attack. If an attacker uses, e.g., a cache-based CSC to communicate or to leak information from a benign process then the cache miss rate will increase significantly. Such changes can be observed by regularly checking key parameters of all running processes. A challenge for a detection mechanism is to detect attacks with few false positives. If applications run in phases then phase transitions can lead to a significant change in the observed behavior as well. A detection mechanism must be able to distinguish between phase changes and attacks. Figure 1 gives an overview of the HexPADS system. HexPADS leverages information from the operating system to collect core process characteristics of all running processes and uses the CPU's PMU to collect detailed low-level performance events from the underlying hardware.

We design HexPADS as a generic process behavior collection mechanism with a plugin-based detection subsystem for different attacks. The core of HexPADS continuously measures a set of parameters for all running processes at negligible overhead. A flexible plugin interface extends the collection mechanism and allows detectors to analyze the behavior of processes. Each plugin detects a certain type of attack using past and current performance data of a process. HexPADS detects

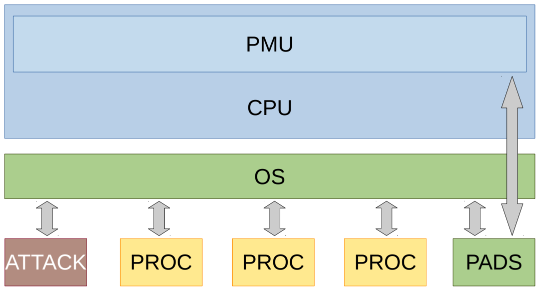

Fig. 1. Overview of the HexPADS system.

attacks by collecting and analyzing system information in five stages that are periodically repeated when the system is running:

1. To gather the necessary runtime information, HexPADS polls detailed process statistics of all running processes. This data is stored in a buffer across iterations to allow aggregate checks, e.g., page faults per iteration. This step takes care of registering new processes (including the setup of performance counters) and cleaning up dead processes.
2. Poll necessary performance counters for each running process. All performance counter results are stored in a buffer to allow aggregate checks and the counters are reset to 0.
3. Calculate performance statistics for each process to, e.g., allow checking if any measured parameter has changed rapidly.
4. Evaluate a set of attack signatures on the measured performance statistics for each process. If an attack signature matches the behavior of a process then a potential attack is detected.
5. If any potential attacks were reported, this step takes evasive or counter measures and reports the attack.

In its default configuration, HexPADS will collect the following performance counters: number of executed instructions, number of last level cache accesses, and number of last level cache misses. In addition to the performance counters, detectors can use the status information of each process as exported from the kernel, e.g., number of minor page faults, number of major page faults, and execution time are used in our signatures. In addition to this baseline, all other information available in the exported process' status can be used and additional counters can be configured. If the amount of desired performance events exceeds the available hardware registers, a time-based sampling scheme can multiplex the available registers (with some loss of precision). HexPADS uses a buffer to store the samples, all elements are initialized with the first measurement.

Attack detectors are functions that evaluate, based on the history of performance samples, if a process is either under attack or attacking another process. If an attack detector matches then it reports the potential attack and the PID to the attack reporting and mitigation module.

Distinguishing between attacking and attacked process is not always straight forward (e.g., a cache-based CSC attack will increase the cache misses in both the attacking and the attacked process). Countermeasures therefore cannot just kill the reported process and other mitigation strategies must be used. Any attack will be reported to the administrator who can decide on specific counter measures. In addition, HexPADS supports a set of automatic counter measures that can mitigate or slow down the attack. HexPADS, e.g., slows down the attacking process (reducing the bandwidth of CSC attacks), stops the attacking process until an administrator can evaluate the situation, or enforces specific scheduling decisions (e.g., pinning processes to disjoint processors[2]). Other mitigation strategies are possible as well, depending on the attack vector.

[2] Scheduling processes on disjoint cores is not enough as the last level cache is shared.

5 Implementation

Following a least privileges principle, HexPADS runs as a user-space daemon and collects information of all running processes. If multiple virtual machines share a single CPU then a HexPADS daemon must run on each VM. Results can then be collected by a central daemon and are evaluated across all running processes on all VMs. Our prototype implementation currently supports monitoring on a single system without distributing the results.

Our prototype follows the design outlined in Section 4 and implements the described analysis loop: it (i) crawls all running processes, updates status information, and initializes performance counters for new processes, (ii) polls the performance counters of all processes, (iii) calculates performance statistics, (iv) evaluates if an attack is in progress, and (v) deploys potential countermeasures against the affected processes. The ringbuffer for the measurements stores the last 60 samples and the scan interval is set to once each second.

The prototype is open-source[3] and the implementation uses less than 2,000 lines of C code. The prototype implementation includes the base framework, detectors for rowhammer, cache CSC attacks, and CAIN attacks and the slow-down and stop the process counter measures. The slow-down counter measure reduces the priority of the identified process and optionally pauses the process to some extend. The stop counter measure stops the process through the `SIG_STOP` signal. We discuss individual detectors in Section 6 as a set of three case studies.

6 Evaluation

Evaluating the performance overhead of HexPADS on a modern system shows that the increased protection results in negligible (non-measurable) performance overhead. Using a set of case studies, we show how HexPADS can detect different attacks: rowhammer attacks, cache CSCs, and CAIN attacks. We have run HexPADS with these detectors on both desktops and servers with regular workload for several days without false positives.

6.1 Performance overhead

The perceived overhead for HexPADS is negligible and makes up for less than 1% of CPU time on a single core on a modern system. To measure impact on other running processes we measured the performance overhead using the SPEC CPU2006 and PARSEC 3.0 benchmarks. We ran our experiment on an Ubuntu 14.04 system with an Intel Core i7-3770 CPU at 3.40 GHz with 4 cores (8 threads), 16 GB of memory. We compiled all SPEC CPU2006 C/C++ benchmarks with clang 3.4 and O3. To reduce noise we averaged over 3 runs using the ref dataset (the default configuration for a reportable run in SPEC CPU2006). We compiled PARSEC 3.0 in its default configuration and evaluate it using the native dataset and 16 threads.

[3] The source code of HexPADS is available at `http://github.com/HexHive/HexPADS`.

SPEC CPU2006	Idle	PADS	Overhead
400.perlbench	306	302	-1.32%
401.bzip2	396	389	-1.80%
403.gcc	242	238	-1.68%
429.mcf	234	211	-10.90%
445.gobmk	374	371	-0.81%
456.hmmer	327	325	-0.62%
458.sjeng	405	403	-0.50%
462.libquantum	287	289	0.69%
464.h264ref	419	417	-0.48%
471.omnetpp	292	292	0.00%
473.astar	304	298	-2.01%
483.xalancbmk	198	197	-0.51%
433.milc	349	334	-4.49%
444.namd	289	288	-0.35%
447.dealII	214	213	-0.47%
450.soplex	195	194	-0.52%
453.povray	126	126	0.00%
470.lbm	200	198	-1.01%
482.sphinx3	400	396	-1.01%
Average	292.47	288.47	-1.39%
Geo.Mean	279.59	275.64	-1.43%

PARSEC	native	PADS	Overhead
blackscholes	36.98	36.93	-0.12%
bodytrack	29.88	30.44	1.88%
canneal	57.06	58.26	2.10%
dedup	13.73	14.02	2.11%
facesim	94.45	96.28	1.94%
ferret	63.64	64.77	1.77%
fluidanimate	72.21	72.40	0.26%
freqmine	81.83	80.88	-1.17%
netdedup	13.04	13.81	5.92%
netferret	407.20	410.16	0.73%
netstr.clust.	132.60	133.32	0.54%
raytace	64.25	65.07	1.27%
streamcluster	121.35	121.93	0.48%
swaptions	45.33	44.95	-0.83%
vips	21.29	21.60	1.47%
x264	17.84	19.48	9.17%
Average	1272.69	1284.30	0.91%
Geo.mean	52.05	52.93	1.69%

Table 1. Performance results for HexPADS on SPEC CPU2006 and PARSEC. Native and HexPADS numbers are in seconds, overhead is in percent.

Table 1 shows the performance results. In general, the overhead for HexPADS is negligible and in our experiment we observed a slight performance improvement for SPEC CPU2006 (likely due to cache variations and fluctuations of the scheduler placing benchmarks on different cores) and a slight performance degradation for PARSEC. The average and geometric mean is less than 2% and therefore likely noise for both benchmarks. The only infrequent false positives we measured were for CAIN on dedup/netdedup (see Section 6.4).

We conducted our experiments on an idle system with multiple cores. The SPEC CPU2006 benchmarks are single threaded but the PARSEC benchmarks are highly parallel. Most of the information is collected by low overhead performance counters and the HexPADS process sleeps most of the time. When observing HexPADS with the `htop` command it uses less than 1% of the CPU to continuously scan, measure, and analyze performance data. In comparison, ninja [7] detects illegal suid processes by scanning the process list at 1.5 - 2% overhead.

6.2 Case study: rowhammer

Rowhammer [25] is a DRAM vulnerability that causes bit flips in DRAM cells, triggered by frequent accesses to neighbouring cells. The DRAM accesses to the adjacent cells cause an interaction with the cell in between, resulting in random bit flips. The rowhammer attack executes cache flush instructions and accesses memory locations in a tight loop. In the attack scenario described by Google's P0 security group, they managed to cause bit flips in a Page Table Entry (PTE) that causes the PTE to point to a physical page under the control of the attacker. This hardware bug allows the attacker to escalate her privileges from user-space to the highest software level, side-stepping all hardware security layers, execution layers, and defense mechanisms.

While incredible powerful, the rowhammer attack is extremely noisy (on the memory bus) and long running. The attack only succeeds if a very large amount of adjacent DRAM accesses are executed in short order, i.e., between refresh intervals that negate all intermediate effects. The attack relies on a high bandwidth to the DRAM cells and therefore has limited interaction with the operating system, e.g., through the page fault handler that adds overhead, reducing the bandwidth for the attack. The overall amount of page faults (or page fault ratio) is therefore low.

```
i ranges from 0 to NR_SAMPLES , not inclusive
cur = current iteration
prev = previous iteration
cache_access = sum(cache_access[i])/NR_SAMPLES
cache_miss = sum(cache_miss[i])/NR_SAMPLES
miss_rate = cache_miss / cache_access
fault_rate = page_faults[cur] / page_faults[prev]
if (
 miss_rate > 0.70 and
 cache_miss > 500,000 and
 fault_rate < 0.01
) cache_attack_detected();
```

Fig. 2. Pseudo code for rowhammer detector based on cache misses and page faults.

Our rowhammer detector (see Figure 2) measures cache accesses and cache misses of all running processes and checks if the cache miss rate is higher than 70% (i.e., more than 70% of all cache accesses are cache misses), the total amount of cache misses is significant, and the number of page faults is low. As rowhammer is a long running attack, our detector averages the cache misses over the sliding window of collected samples. In addition, the average page table miss rate must be low, otherwise the memory accesses would not happen fast enough. If the cache miss rate is too low then no bits are flipped. Using the rowhammer prototype implementation[4] we always measured a cache miss rate of $> 90\%$ (more than 4,000,000 cache misses per iteration, the highest number of cache misses of a benign process was 101,000 cache misses per iteration) and the attack is

[4] Google's prototype implementation is available at `https://github.com/google/rowhammer-test`.

detected immediately after the process starts up. Any successful rowhammer attack will always be noisy and the cache miss rate per instruction must be high for the attack to be successful. The default counter measure slows down an offending process for a configurable amount of time.

6.3 Case study: cache-based CSCs

Cache CSCs are very similar in their cache access patterns to the rowhammer attack. Generally, a cache CSC uses one of three ways to communicate [8]: (i) evict and time (the attacker measures execution of the victim's code, evicts the cache, and measures the same code again), (ii) prime and probe (the attacker fills its own memory and measures through access times what data was evicted by the victim), or (iii) flush and reload (the attacker flushes shared memory and measures what memory was reloaded by the victim). All these attacks have in common that they result in a huge amount of cache misses in a short amount of time as large memory areas have to be flushed and read/written.

We have tested two cache covert channels: (i) cache template attacks [11] which is based on flush and reload and (ii) an enhanced version of C5 [18] which is based on prime and probe. The observed memory access pattern is very similar to rowhammer attacks with the difference that a cache CSC is only concerned about the cache itself and not if the memory is written back to DRAM. In our experiment, cache template attacks results in about 1,500,000 cache misses per iteration and C5 attacks in about 2,300,000 cache misses per iteration.

We therefore use the same detector as for rowhammer to detect cache CSCs. The covert channels described above rely on a combination of repeated flushing or filling of the cache and measuring timing. The cache flushing and filling is measurable through cache misses, indicating that a cache CSC is being used.

Our current detector does not distinguish between rowhammer and cache CSCs and successfully detects both attacks. Cache CSCs will always incur a high amount of cache misses, just like rowhammer attacks. If the attacker lowers the speed of the cache attack, the bandwidth will decrease alongside which results in additional noise on the channel. After a certain noise level is reached the attack becomes unrealistic.

6.4 Case study: CAIN

CAIN (Cross VM ASL INtrospection) [2] leverages memory deduplication as a side channel to recover ASLR base addresses of loaded libraries in co-located virtual machines. For a successful attack, an attacker needs to execute user-space code on a virtual machine that is co-located with the target machine (i.e., runs on the same physical hardware). Memory deduplication searches for shared memory pages across virtual machines and coalesces any common pages. A write to a merged page results in a page fault caught by the VMM and triggers a copy-on-write operation, resulting in a timing side channel that allows the detection of specific memory pages in concurrently running virtual machines. Memory deduplication saves physical memory but causes performance degradation when pages are unmerged (e.g., when one virtual machine writes to the page). CAIN generates a large amount of page candidates for specific libraries, picking a page that is static except for a set of pointers relative to the library's base address. Each generated page candidate then has the probability of $\frac{1}{\text{ASL entropy}}$ of being present in the target virtual machine. CAIN uses all available memory to generate target pages and then waits for the memory deduplication mechanism to merge a candidate page and the target page. The correct target page is then detected by measuring timing when writing to the page (due to the copy-on-write it takes a longer time to write compared to an unmerged page).

CAIN behavior is naturally bursty and generates a large amount of page faults and cache misses in a short time whenever new candidate pages are generated. This behavior is easily detected by measuring the gradient of page faults and the amount of cache misses (for writing).

```
cur = current iteration
prev = previous iteration
page_faults = array of page fault measurements
cache_miss = array of cache miss measurements
page_miss_rate = page_faults[cur]/executed_instr

if (
 page_faults[prev] > 2.0 * page_faults[cur] and
 page_faults[cur] > 100000 and
 cache_miss[cur] > 10000 and
 page_miss_rate > 0.001
) CAIN_attack_detected();
if (
 page_faults[prev] + page_faults[cur] > 256000
) CAIN_attack2_detected();
```

Fig. 3. Pseudo code for CAIN detector based on cache misses and page faults.

Our detector (see Figure 3) checks if (i) the amount of page faults in the current iteration is more than double the amount of page faults in the previous iteration (i.e., the amount of page faults doubled), there were more than 100,000 page faults, more than 10,000 cache misses in this iteration, and the page miss rate per executed instruction in the last interval was higher than 0.001 or (ii) the amount of page faults in the last two iteration is higher than 256,000 (which corresponds to 1024MB of memory being initialized in a short interval). The first part of the detector checks the increasing flank while the second part checks for a high amount of new memory that is allocated in a short burst. Our detector currently does not check for the ratio between read and write cache misses, for CAIN the amount of write cache misses would be much higher than the amount of read cache misses. Only the PARSEC dedup/netdeup benchmarks experienced false positives as this benchmark allocates a huge amount of memory during startup. For the complete evaluation, the first check results in 1 false positive and the second check in 24 false positives. CAIN attacks are not time critical, so for a future detector we will ensure that benign cases that continuously use the allocated memory do not trigger a detection.

The current detector measures the memory allocation pattern of a CAIN attack through page faults, cache misses, and the amount of allocated memory. CAIN attacks could mitigate the detection by allocating less memory, which would reduce the effectiveness of the attack. An extension of the detector could measure the absence of accesses after detection to detect the phase where CAIN is waiting for the VMM to merge individual pages.

6.5 Discussion, limitations, and future work

The efficiency and success of HexPADS depends on the ability of the detectors to distinguish benign behavior from malicious behavior. The attacks evaluated in the case studies are fundamentally different from benign applications due to the underlying constraints of the attacks. With knowledge of the signatures (which will likely be widely distributed and analyzed), an attacker could launch some form of targeted Mimicry [30] attacks. Mimicry attacks hide the malicious behavior in benign behavior, thereby circumventing detection. HexPADS is not immune to Mimicry attacks and an attacker could, e.g., slow down the number of cache accesses to evade the rowhammer detection. But by slowing down the attack it becomes less efficient and more likely to fail, e.g., for rowhammer, if the attack does not achieve a sufficiently high number of memory accesses between memory refresh operations then the attack will fail. The design of effective detectors depends on a threshold where the attack is no longer successful, yet the amount of false positives remains low. We acknowledge the difficulty of finding such efficient thresholds, especially for programs with different program characteristics where the threshold must be conservative.

In the current version, the baseline behavior and the signatures are hardcoded. The current signatures are based on manual analysis of program executions. As future work we will look into ways of coming up with tighter and more precise signatures automatically, e.g., by collecting benign traces of a wide variety of applications and workloads and using machine learning to automatically extract a baseline pattern and classify the different samples into general signatures. In addition, we will look into aggregating performance measurements of child processes to mitigate an attacker that constantly spawns children to prevent detection. The current motivating examples and case-studies focus on memory attacks. In future work, HexPADS can either be extended to include other attack vectors (e.g., by sampling other performance events), or its concept can be integrated into other attack detection frameworks.

The current prototype implementation is limited to single host detection and does not coordinate information across different virtual machines (i.e., the detection mechanism must run on the same virtual machine as the attacker). This is merely an engineering limitation and the prototype can be extended through additional programming effort. A CAIN attack can only be observed on the same system, so the detector must either run on the attacker machine (e.g., in the case where the attacker controls only a user-space application) or at the level of the hypervisor. An advantage of the current implementation is that the daemon has negligible overhead and runs without any elevated privileges. Disadvantages of such an implementation are that (i) only effects on the system can be observed, attacks from non-monitored systems (virtual machines) are missed and (ii) the operating system is a part of the trusted computing base, any attacker with elevated privileges (administrator privileges) can disable the monitoring and detection mechanism.

7 Related work

Related work for HexPADS exists in different areas. On one hand, prior work on CSC attacks is used as a motivation to develop our attack detection mechanism and we use different CSC mechanisms to evaluate our work. On the other hand, we compare our work against different existing CSC attack detection and mitigation mechanisms, showing key differences between our performance counter based approach and other approaches that focus on mitigation instead of detection. Last but not least, we compare against other existing intrusion detection mechanisms and explain why they detect attacks on a different abstraction level.

7.1 Covert and Side channel attacks

Last level caches are a prime target to extract information using CSC information leaks across processes or even across virtual machines. Sensitive information (e.g., cryptographic keys) can be extracted from unwilling sensitive processes [24,39,37,11] or two malicious processes can use the covert channel to communicate stealthily [18]. A challenge for these CSC attacks is the underlying hardware configuration as each CPU family can be different. Unfortunately, an automated exploration of the cache configuration is possible [19,11].

Other CSCs include, e.g., the last branch target buffer [1], the memory bus [36], memory deduplication mechanisms [26,14,2], and attacks against the underlying memory architecture [25].

7.2 Covert and side channel attack detection and mitigation

A CSC attack detection mechanism may be implemented at the level of the hardware, the virtual machine monitor, the operating-system, or the application.

Hardware-based detection and mitigation mechanisms can be separated into approaches that partition resources [6,31,32] with the downside of potentially under-utilizing resources, randomizing accesses [32,33], or limiting the granularity of the timer [17].

On the hypervisor level, HomeAlone [38] detects cross-VM side channel attacks by monitoring cache misses and cache behavior. Other defense mechanisms in the hypervisor either partition resources to be used exclusively for a given virtual machine [15] (with the drawback that same-machine attacks are possible) or limit the timer granularity for virtual machines [28]. HexPADS in comparison measures fine-grained performance events on the process level and allows the identification of individual processes that cause the outlier.

Düppel [40] employs periodic cache flushing to introduce noise and to reduce the attacker's bandwidth. This is a pure mitigation mechanism that does not distinguish between benign behavior and attack behavior. HexPADS may use a mechanism to mitigate an ongoing attack as soon as it is detected with the advantage that cache flushing (and the associated overhead) only occurs during active attacks and not whenever a sensitive operation is executed.

7.3 Intrusion detection and mitigation

Network-based IDS like Bro [22] detect an intrusion by inspecting network packets. Host-based IDS observe system characteristics like system call patterns and parameters [13,34,20], log analysis [3], or file integrity checking (e.g., AFICK, Tripwire, or AIDE [10,3]) to detect malicious activity. Intrusion detection systems are either misuse-based or anomaly-based. A misuse-based IDS matches a set of patterns against the observed pattern [23,22,29]. An anomaly-based IDS detects deviations from a well known, good baseline [5,16,9,20,35,7].

HexPADS targets microarchitectural features and uses performance counters to collect fine-grained system information to detect attacks that are not directly observable by regular introspection methods but need support from hardware performance monitors (e.g., by measuring the amount of cache misses).

8 Conclusion

Intrusion detection and attack detection systems enable the detection of otherwise uncaught attacks (i.e., if all other defense mechanisms fail). We have presented the design and open-source implementation of HexPADS, a novel attack detection mechanism that leverages both core systems parameters and performance counter-based statistics on program execution to detect ongoing attacks. The core system measures a set of system parameters and performance characteristics like, e.g., cache misses, executed instructions, or page faults. Through a flexible plugin mechanism we can add dynamic detectors for individual attacks. In three case studies we have evaluated HexPADS and shown its effectiveness against rowhammer, cache-based covert and side channels, and CAIN attacks by implementing simple detectors that use cache accesses, cache misses, page faults, and number of executed instructions to detect attacks. The performance overhead of HexPADS is negligible (non-measurable) and the flexible design and plugin structure simplifies adding new detectors for other and future attacks.

9 Acknowledgments

We would like to thank Clémentine Maurice, Daniel Grauss, Antonio Barresi, Scott A. Carr, and Terry Ching-Hsiang Hsu for generous feedback on the paper. We also thank Clémentine and Daniel for providing access to the CSC implementation and Antonio for providing access to the CAIN implementation. This work was sponsored, in part, by NSF CNS-1513783.

References

1. O. Acıiçmez, c. K. Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *RSA Conference on Topics in Cryptology*, CT-RSA'07, 2006.
2. A. Barresi, K. Razavi, M. Payer, and T. R. Gross. CAIN: Silently Breaking ASLR in the Cloud. In *WOOT'15: 9th Usenix Workshop on Offensive Technologies*, 2015. (35% acceptance rate – 20/57).
3. D. B. Cid. Ossec: Open source host-based intrusion detection system. `http://ossec-docs.readthedocs.org/en/latest/`, 2015.
4. I. Corp. Intel 64 and IA-32 Intel Architecture Software Developer's Manual Combined Volumes 3A and 3B: System Programming Guide, Parts 1 and 2, 2015.
5. D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 1987.
6. L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 2012.
7. T. R. Flo. ninja process monitor. `http://forkbomb.org/ninja/`, 2010.
8. A. Fogh. Cache side channel attacks. `http://dreamsofastone.blogspot.com/2015/09/cache-side-channel-attacks.html`, 2015.
9. A. Ghosh, J. Wanken, and F. Charron. Detecting anomalous and unknown intrusions against programs. In *Annual Computer Security Applications Conf.*, 1998.
10. L. Grim and R. Vandenbrink. Ids: File integrity checking. Technical report, SANS Institute, 2014.
11. D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, 2015.
12. E. Hiroaki and Y. Kunikazu. ProPolice: Improved stack-smashing attack detection. *IPSJ SIG Notes*, pages 181–188, 2001.
13. S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3), 1998.
14. G. Irazoqui, M. Inci, T. Eisenbarth, and B. Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In *Intl. Symp. on Research in Attacks, Intrusions, and Defenses*, 2014.
15. T. Kim, M. Peinado, and G. Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security Symposium*, 2012.
16. C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. In *IEEE Symp. on Security and Privacy*, 1997.
17. R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Intl. Symp. on Computer Architecture*, 2012.

18. C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: Cross-cores cache covert channel. In *Conf. on Detection of Intrusions and Malware and Vulnerability Assessment*, 2015.
19. C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Intl. Symp. on Research in Attacks, Intrusions, and Defenses*, 2015.
20. D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1), 2006.
21. PaX-Team. PaX ASLR (Address Space Layout Randomization). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.
22. V. Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999.
23. P. A. Porras and P. G. Neumann. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *In Proceedings of the 20th National Information Systems Security Conference*, 1997.
24. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conf. on Computer and Communication Security*, 2009.
25. M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. `http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`, 2015.
26. K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Memory Deduplication As a Threat to the Guest OS. In *European Workshop on System Security*, 2011.
27. A. van de Ven and I. Molnar. Exec shield. `https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.
28. B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine-grained timers in xen. In *ACM Cloud Computing Security Workshop*, 2011.
29. G. Vigna, F. Valeur, and R. A. Kemmerer. Designing and implementing a family of intrusion detection systems. In *European Software Engineering Conference*, 2003.
30. D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM Conf. on Computer and Communication Security*, 2002.
31. Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *Annual Computer Security Applications Conf.*, 2006.
32. Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Intl. Symp. on Computer Architecture*, 2007.
33. Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Intl. Symposium on Microarchitecture*, 2008.
34. C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusion using system calls: alternative data models. In *IEEE Symp. on Security and Privacy*, 1999.
35. J. Wu, L. Ding, Y. Wu, N. Min-Allah, S. U. Khan, and Y. Wang. c^2 detector: a covert channel detection framework in cloud computing. *Security and Communication Networks*, 7(3), 2014.
36. Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security Symposium*, 2012.
37. Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, 2014.
38. Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *IEEE Symp. on Security and Privacy*, 2012.

39. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *ACM Conf. on Computer and Communication Security*, 2012.
40. Y. Zhang and M. K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side-channels in the cloud. In *ACM Conf. on Computer and Communication Security*, 2013.