# Real time detection of cache-based side-channel attacks using Hardware Performance Counters

Marco Chiappetta
*Faculty of Engineering and Natural Sciences*
*Sabanci University*
*Istanbul, Turkey*
*Email: marcoc@sabanciuniv.edu*

Erkay Savas
*Faculty of Engineering and Natural Sciences*
*Sabanci University*
*Istanbul, Turkey*
*Email: erkays@sabanciuniv.edu*

Cemal Yilmaz
*Faculty of Engineering and Natural Sciences*
*Sabanci University*
*Istanbul, Turkey*
*Email: cyilmaz@sabanciuniv.edu*

***Abstract*—In this paper we analyze three methods to detect cache-based side-channel attacks in real time, preventing or limiting the amount of leaked information. Two of the three methods are based on machine learning techniques and all the three of them can successfully detect an attacker in about one fifth of the time required to complete the attack. There were no false positives in our test environment. Moreover we could not measure a change in the execution time of the processes involved in the attack, meaning there is no perceivable overhead. We also analyze how the detection systems behave with a modified version of one of the spy processes. With some optimization we are confident these systems can be used in real world scenarios.**

## 1. Introduction

Side-channel attacks are a particular class of attacks, usually targeting cryptographic algorithms, which do not exploit a flaw in the design of the algorithms themselves but rather in their implementation.

Cache-based side-channel attacks represent a subset whose purpose is to retrieve sensitive information from a system just by exploiting the shared cache memory in modern CPUs [1]. Moreover such attacks can be conducted between virtually isolated environments such as virtual machines or Linux containers.

As described in the next section a class of cache-based attacks rely on the presence of a userland assembly instruction to partially or fully manipulate the state of the shared cache (in the case of Intel CPUs the instruction is `CLFLUSH`) and the presence of a feature, such as KSM [2] or TPS [3], which allows processes to share identical pages in memory.

To prevent such attacks between processes or virtual machines we would either need to switch to a CPU architecture that prevents the usage of the aforementioned instruction or to disable any memory optimization feature. In the first case it would be necessary to recompile any incompatible program for the new architecture (e.g. ARM) while in the second case there would be a loss of performance given by the fact that processes would be unable to share identical pages, therefore increasing memory consumption.

With regard to virtual machines another problem is colocation. That is, to carry out the attack it is necessary that the attacker's virtual machine and the victim's virtual machine run on the same physical hardware, therefore sharing the main memory and the cache. Such problem was partially solved by Ristenpart et al. [4] who were able to colocate two virtual machines on the Amazon EC2 cloud computing service with a probability of 40%.

The first practical implementation of a cache-based attack was presented by Tsunoo et al. in [5] and targeted the DES algorithm. In [6] Osvik et al. devised two techniques (`EVICT+PRIME` and the more efficient `PRIME+PROBE`) to attack AES by evicting everything in the cache and measuring the time for an encryption. More recent cache attacks include [7], by Yarom and Falkner, that uses the `FLUSH+RELOAD` technique to retrieve the private exponent used in GNU Privacy Guard (GPG)'s implementation of RSA, [8] by Yarom and Benger where the same technique is used against the ECDSA implementation in `OpenSSL` and [9], by Irazoqui et al., and [10], by Gulmezoglu et al., where `FLUSH+RELOAD` is used to detect the key used in the last round of an AES encryption.

The problem we address is to detect such attacks in time, before they are complete, to be able take the proper countermeasures, i.e. to kill the suspicious process, in a same-OS scenario, or relocate the virtual machine, in a cross-VM scenario.

In this paper we present three methods, of which two are based on machine learning techniques, that can be combined or used separately to detect cache-based side-channel attacks at runtime, with a particular focus on those using the FLUSH+RELOAD technique [7]. Our methods do not require any modification to the operating system and run as normal user-level processes. The only requirement is the availability of hardware performance counters, quite common on most modern CPUs [20].

The paper is organized as follows: in Section 2 we present necessary background information on cache-based side-channel attacks and hardware performance counters

followed by an analysis of three attacks against RSA, AES and ECDSA. In Section 3 we describe our methods and their advantages and shortcomings. In Section 4 we show our results and how it is possible to detect an attack in time to take the proper countermeasures; we did not experience any change in the execution time of the processes involved in the attack both while and while not using our detection methods showing that our tools are able to detect an attack with virtually no overhead. Section 5 presents an improved version of one of the attacks that is able to deceive the first (and simplest) detection method while still being able to complete an attack, although in more time. We believe this might trigger interest in further research on how to deceive, and therefore improve, detection systems for this kind of attacks. Sections 6 and 7 present a discussion about our results, and their implications, and the feasibility of employing such detection systems in real world scenarios.

## 2. Background

### 2.1. Cache-based side-channel attacks

Numerous attacks based on shared hardware and software resources have been carried in the past.

Recently those based on CPU's cache memory turned out to be very effective, easy to implement and fast. This paper focuses on a particular class of cache-based side-channel attacks that utilize a technique named FLUSH+RELOAD [7].

The entities involved in this attack are usually two processes: a victim and a spy. The victim performs some kind of cryptographic operation (i.e. encryption, decryption or signature) where some secret data, likely a key, is being used while the spy attempts to capture such data by analyzing the victim's behavior.

The success of the attack mainly depends on three factors: the ability of the spy to synchronize with the victim (that is, start the attack as soon as the cryptographic operation starts), the presence of a user-level instruction to evict a specific area of the CPU's cache and the presence of mechanisms like Transparent Page Sharing (TPS) [3] or Kernel Same-page Merging (KSM) [2].

KSM was implemented for the first time in Linux 2.6.32 as a technique to augment memory density and it is enabled by default. It allows processes to transparently share identical pages by mapping addresses which belong to different virtual addresses to the same physical address. Two downsides of KSM are the high CPU load needed to regularly run the merging process [12] and the fact that it makes attacks like FLUSH+RELOAD feasible.

TPS is, instead, a proprietary technology of VMWare whose purpose is to make virtual machines share identical pages with the hypervisor taking care of looking for and merging them. The feature is enabled by default in both their cloud and desktop solutions until the latest version (6.0 at the time of writing) [13] where it has to be manually enabled because of security concerns [14].

Since two merged pages are mapped to the same physical address, in the main memory, different processes that try to retrieve a shared page cause the MMU to access the same physical address. Furthermore the cache is mapped onto the same address space and content that is evicted from it will be evicted for all processes that share it in the main memory.

Another important requirement for the attack to be feasible is the existence of a user level instruction that allows to evict a specific address from the CPU's cache. On most modern Intel processors (mainly Core i3, i5, i7 and Xeon) such instruction exists under the assembly mnemonic CLFLUSH [15].

Calling CLFLUSH with a single address causes the whole cache line, which includes the content from that address, to be evicted. Furthermore, on Intel CPUs, cache levels form an inclusive hierarchy: the L3 cache includes the L2 cache content and the L2 cache includes the L1 cache content. For this reason evicting a line from the LLC (L3) propagates the eviction to the lower levels as well. The algorithm roughly works as in Example 1.

**Example 1.** Algorithm for a generic FLUSH+RELOAD attack.

```
Assume 0xABC to be a physical address in a page
shared by the spy and the victim.
Repeat until victim terminates:
1. Victim accesses 0xABC.
2. Spy evicts 0xABC from the LLC and sleeps
   for a few clock cycles (to be determined
   according to the victim process).
3. Victim may or may not access 0xABC.
4. Spy loads 0xABC and keeps track of how
   long it takes.
5. If it takes longer than a specific threshold
   it means the victim did not access 0xABC and
   therefore it was not in the cache.
   Else the victim accessed 0xABC and it was
   put in cache.
```

The original FLUSH+RELOAD attack [7], by Yarom et al., focused on guessing which instructions are being executed by the victim. In fact, by knowing which instructions are or are not executed while performing a cryptographic operation it is often possible to retrieve information that can be used to reconstruct the secret used during the process, such as encryption keys.

In particular the first attack of this kind was used to determine the bits of the key used in GPG when performing a decryption with RSA, thanks to a vulnerable implementation of the square multiply algorithm.

To address the concern that this attack would only work on non-constant-time implementations (that is, implementations whose execution time highly depends on the input, especially sensitive input like encryption keys, that determine which instructions are executed) a second version of the attack was released, this time aimed at breaking the supposedly robust implementation of the point scalar multiplication algorithm based on the Montgomery ladder used in OpenSSL's ECDSA [8].

A variant of such technique [9], i.e. the third type of attack, by Irazoqui et al. was able to retrieve all the key bits by observing a few seconds to a minute worth of AES encryptions or decryptions, although the amount of time required to complete an attack makes it more prone to be detected as demonstrated by our experiments.

In our work we aimed at detecting the second and third types of attack, the first being a simpler version of the second.

## 2.2. Attacking RSA

While performing a signature or a decryption with RSA there is the need to compute $m^d mod n$ where $m$ is the plaintext, $d$ is the private exponent and $n$ is the product of two large prime. One algorithm to perform such computation is square multiply, also known as binary exponentiation, described in Algorithm 1.

**Algorithm 1** Square multiply

```
1:  procedure SQUARE-MULTIPLY(m, d, n)
2:      x = 1
3:      for i in bin(d) do
4:          x = x^2
5:          x = x mod n
6:          if i == 1 then
7:              x = x * m
8:              x = x mod n
9:          end if
10:     end for
11:     return x
12: end procedure
```

Given the non-constant time nature of the algorithm its implementations are vulnerable to different kinds of side-channel attacks, including those based on timing and power analysis [16]. In particular the operations performed according to the value of each bit leak valuable information that can be used to reconstruct them.

In [7] Yarom et al. exploited the implementation of the square multiply algorithm inside GNU Privacy Guard (GPG). By reverse engineering the `OpenSSL` binary it is possible to retrieve the memory addresses of lines 7 or 8, from Algorithm 1, in the GPG binary and use them to carry the attack described in Algorithm 2.

The attack briefly works as follows: the spy starts executing a loop in which it first flushes the targeted address, in line 4, then waits an empirically determined number of cycles before reloading the address, in line 6. When the spy reloads the address there exist two possibilities corresponding to the two branches of the conditional at line 7: if the victim accessed its content the loading time will be lower than a predefined threshold, which means the bit was likely 1, otherwise it will be higher, which means the bit was likely 0.

In [7] the authors reported that, on average, the percentage of the private exponent's bits that can be recovered is 96.7% with a worst case of 90%, in a cross-VM scenario, and 98.7% with a worst case of 95% on the same operating system.

**Algorithm 2** FLUSH+RELOAD attack on RSA

```
1:  procedure FLUSH-RELOAD-RSA(addr)
2:      bits = [ ]
3:      while True do
4:          flush(addr)
5:          sleep(ncycles)        ▷ ncycles empirical value
6:          t = reload(addr)                    ▷ t1 = time
7:          if t1 < threshold then
8:              bits.append(1)
9:          else
10:             bits.append(0)
11:         end if
12:     end while
13:     return bits
14: end procedure
```

## 2.3. Attacking AES

Irazoqui et al. devised an algorithm to fully recover the scheduled key used in the last round of AES in a matter of seconds to minutes [9]. Their algorithm is a variant of the more generic FLUSH+RELOAD that focuses on guessing which values of the AES lookup tables were accessed and uses this information to reconstruct the key.

The lookup tables in AES allow to speed up the SubBytes, ShiftRows and, except for the last round, MixColumns steps turning them into a single lookup operation plus a XOR operation to obtain the final ciphertext. In a byte-oriented implementation of AES the i-th ciphertext byte is therefore produced as follows:

$$C_i = T[S_i] \text{ XOR } K_i.$$

where $T$ is the lookup table, $S_i$ is the i-th byte of the current state, used as an index for $T$, and $K_i$ is the i-th byte of the key.

$$\begin{aligned} C_{00} &= 0x35, 0x87, 0x65, \mathbf{0xfa} \\ C_{01} &= 0x21, 0x10, \mathbf{0xfa}, 0x61 \\ C_{02} &= \mathbf{0xfa}, 0xa1, 0xa9, 0x45 \\ C_{03} &= 0x01, \mathbf{0xfa}, 0xc4, 0xf5 \end{aligned}$$

Figure 1. Combinations of ciphertext and T-table bytes that highlight the last round's key used in AES, $0xfa$ in this case.

Without loss of generality let us analyze the first table $T_0$. For AES-128 each table entry holds 4 bytes, hence $T_0 = \{T_{00}, T_{01}, T_{02}, T_{03}\}$. Therefore it can produce 4 possible values for the first byte of the ciphertext, hence $C_0 = \{C_{00}, C_{01}, C_{02}, C_{03}\}$. By performing a XOR operation between all possible pairs of $T_{ij}$ and $C_{ij}$ we obtain 4 sets of 4 bytes each. These 4 sets will have a value in common, as shown in Figure 1, which will be the value of the first byte of the round key.

Assuming a cache line holds all 4 bytes of $T_0$ we can monitor such line and check whether the table was accessed, in which case we can perform the aforementioned operations to discover the key.

## 2.4. Attacking ECDSA

A message signed with ECDSA consists of a triple $(m, r, s)$ where $m$ is the message and $r$ and $s$ are computed as in Algorithm 3. We assume a group of order $n$ and that $G$ is a generator of such group. Specifically the curve used in the attack is `sect571r1` whose parameters are described in [17].

The ephemeral key $k$ used in the signature algorithm can be exploited to retrieve the private key $d$ since $d = (sk - z)r^{-1}$ and $s$, $z$ and $r$ are known (see Algorithm 3).

Attacking an implementation of the signature algorithm means, indeed, attacking the step where the point $(x, y)$ is computed, as shown in Step 4 of Algorithm 3. In fact the implementation of the point multiplication algorithm used for the computation can lead to some data leakage that provides information for an attacker to reconstruct the ephemeral key.

**Algorithm 3** ECDSA signature

1: **procedure** SIGNMESSAGE($m, G$) ▷ $m$ = message
2: $z = truncate(hash(m), L_n)$
3: $k = random(1, n-1)$
4: $(x, y) = k * G$ ▷ $G$ = generator
5: $r = x \bmod n$
6: $s = k^{-1} * (z + r * d) \bmod n$ ▷ $d$ = private key
7: **return** $(m, r, s)$
8: **end procedure**

A simple implementation of the point multiplication algorithm, called double-and-add, is provided in Algorithm 4. Such implementation could be exploited with the same process shown in the previous section.

**Algorithm 4** Double-and-add point scalar multiplication

1: **procedure** DOUBLE-AND-ADD($k, P$)
2: $Q = P$
3: **for** $i$ in $bin(k)$ **do**
4: $double(Q)$ ▷ $Q = 2Q$
5: **if** $i == 0$ **then**
6: $add(Q, P)$ ▷ $Q = Q + P$
7: **end if**
8: **end for**
9: **return** $Q$
10: **end procedure**

In fact it can be noticed that by using a simple attack based on `FLUSH+RELOAD` we can guess when a bit is 0 or 1 by monitoring the cache line corresponding to the function called at Step 6 in Algorithm 4. Whenever the bit is 0 the line will be loaded in cache by the victim and the loading time in the spy will be shorter otherwise it is be fair to assume the bit is 1.

To avoid this kind of attacks OpenSSL uses a different implementation based on the Montgomery ladder [18]. Algorithm 5 provides an example. The Montgomery ladder relies on the same functions being called regardless of whether the bit is clear or set. The only change between the two cases is in the order of the arguments passed to the functions.

**Algorithm 5** Montgomery ladder point scalar mulitplication

1: **procedure** MONTGOMERY-LADDER($k, P$)
2: $R_0 = 0$
3: $R_1 = P$
4: **for** $i$ in bin($k$) **do**
5: **if** $i == 0$ **then**
6: $add(R_1, R_0)$ ▷ $R_1 = R_0 + R_1$
7: $double(R_0)$ ▷ $R_0 = 2R_0$
8: **else**
9: $add(R_0, R_1)$ ▷ $R_1 = R_0 + R_1$
10: $double(R_1)$ ▷ $R_1 = 2R_1$
11: **end if**
12: **end for**
13: **return** $R_0$
14: **end procedure**

OpenSSL's implementation was broken by Yarom et al. [8] proving that `FLUSH+RELOAD` can be used even when the algorithm is supposed to resist against timing attacks. The target of the attack is the code contained in function `ec_GF2m_montgomery_point_multiply`, a sample of which is shown in 2.

To perform the point scalar multiplication using the Montgomery ladder the scalar $k$ is read bit by bit in a loop. According to the value of each bit a different conditional branch is taken and the functions to add and double the point, `gf2m_Madd` and `gf2m_Mdouble`, are called with the arguments in a different order. The principle behind this design is that since the same functions are called regardless of the state of the current bit, an attack based on timing would fail.

With some reverse engineering on the OpenSSL binary it is possible to retrieve the memory addresses of the lines of interest: 275, 276, 280 and 281. Because of spatial prefetching it is necessary to probe addresses that are as distant as possible from each other in memory (and consequently in the cache).

Specifically lines 275 and 281 of Figure 2, whose memory addresses are passed as arguments *addr1* and *addr2* in Algorithm 6, are suitable for the attack since they lie at the very beginning and the very end of the main conditional branch of line 273. The attack proceeds by flushing and reloading these addresses to understand which ones were accessed.

Assuming the victim starts executing the loop and at the same time the spy starts the main loop in Algorithm 5 the two processes are perfectly synchronized and the attack has the highest likelihood of success.

It is necessary, for the spy, to sleep for a certain amount of CPU cycles ($n_{cycles}$ in Algorithm 6) equal to the average

```
268 for (; i >= 0; i--)
269 {
270 word = scalar->d[i];
271 while (mask)
272     {
273     if (word & mask)
274         {
275         if (!gf2m_Madd(
          group, &point->X, x1, z1,
          x2, z2,
          ctx))
          goto err;
276         if (!gf2m_Mdouble(group,
x2, z2,
          ctx))
          goto err;
277         }
278     else
279         {
280         if (!gf2m_Madd
          (group, &point->X, x2, z2,
          x1, z1,
          ctx))
          goto err;
281         if (!gf2m_Mdouble(group,
x1, z1,
          ctx))
          goto err;
282       }
283     mask >>= 1;
284     }
285 mask = BN_TBIT;
286 }
```

Figure 2. Main loop of the Montgomery ladder implementation in OpenSSL

number of cycles needed for the victim to complete a loop. The actual time is not always constant but depends on how the process is scheduled. For example the spy might be scheduled such that its loop is executed twice even though the victim's is executed only once, in which case the second measurement would be ignored and the spy would wait for the next iteration.

In line 10 of Algorithm 6 other than just checking whether $t_1 < \tau$ we also check whether $t_2 > \tau$. That is, we make sure that $addr1$ was not loaded in memory because of spatial prefetching [19]. With spatial prefetching if two addresses belong to the same set of lines loaded from the main memory it is impossible to understand which one was intentionally loaded by a process and which one was retrieved because of this feature. In this case we want to make sure that $addr1$ was not loaded in the cache just because $addr2$ was (and viceversa in line 12).

Another issue arises when the spy has to terminate. If it terminates too soon it will miss some of the last bits so the best course of action would be to keep executing the loop

**Algorithm 6** FLUSH+RELOAD attack on ECDSA

```
1: procedure FLUSH-RELOAD-ECDSA(addr1, addr2)
2:     bits = [ ]
3:     τ = getthreshold()
4:     while True do
5:         flush(addr1)
6:         flush(addr2)
7:         sleep(n_cycles)        ▷ wait for a loop to complete
8:         t1 = reload(addr1)                    ▷ t1 = time
9:         t2 = reload(addr2)                    ▷ t2 = time
10:        if t1 < τ and t2 > τ then
11:            bits.append(1)
12:        else if t2 < τ and t1 > τ then
13:            bits.append(0)
14:        end if
15:    end while
16:    return bits
17: end procedure
```

until a certain number of bits equal to 0 (i.e. both $t1$ and $t2$ are above the threshold) is reached.

At the end of the attack some (or all) of the bits of the ephemeral key are recovered and it is possible to reconstruct the private key. In the worst case the attack is known to miss 34 bits but the actual value of the scalar $k$ can be restored, by using the baby step giant step algorithm, in less than one second of computation and using just 10 MB of memory [8].

## 2.5. Hardware Performance Counters

Modern microprocessors are equipped with special purpose registers used to store data about a wide range of CPU related events: clock cycles, cache hits, cache misses, branch misses etc. Such registers, called Hardware Performance Counters (in short HPCs), are commonly used to profile the behavior of a program and understand what to optimize in order to increase its performance [20]. In this paper we describe an alternative usage of such feature that allows us to collect predictive data about one or more processes with little overhead.

Similar alternative usages are described in [24] where the timing function of a particular time based cache attack is replaced with data coming from HPCs, [25] where exploits are detected by constructing a dynamic signature of the processes involved and [26] which briefly mentions how it would be possible to mitigate the cache-based side-channel attacks described in this paper through the use of HPCs.

The Linux kernel, assuming the target CPU supports them, provides an interactive interface to the HPCs via a command-line tool named `perf` [21]. The tool allows to collect, visualize, filter and aggregate data gathered through the HPCs on a system-wide, process or even thread basis.

The most interesting sub-command, for the purposes of our experiments, is `perf-stat`. Using this utility it is possible to specify which events to monitor, a target process or thread, the output format and the interval of time

```
$ perf stat  make -j

Performance counter stats for 'make -j':

8117.370256  task clock ticks
       678  context switches
       133  CPU migrations
    235724  pagefaults
24821162526  CPU cycles
18687303457  instructions
 172158895  cache references
  27075259  cache misses

Wall-clock time elapsed:   719.554352 msecs
```

Figure 3. Sample output of the perf-stat utility

between two consecutive reports. An example report from `perf-stat`, while monitoring the execution of the tool make, is shown in Figure 3.

An important shortcoming of `perf-stat` is its limited resolution; `perf-stat` gives the opportunity to sample HPCs multiple times in a second but the minimum interval between two consecutive samples is 100 ms.

During the experiments we observed that the average time to perform a signature with OpenSSL, using the ECDSA algorithm with curve `sect571r1`, is 6 ms on the testing system. Since one of our aims is to be able to detect even the fastest implementation of the FLUSH+RELOAD attack (which only needs a single signature round to succeed) it becomes obvious that the resolution of `perf-stat` was insufficient.

We developed a custom utility, called `quickhpc` [33] that offers a subset of the features of `perf-stat` but with some improvements.

The tool `quickhpc` can be run as a normal user level process and requires the privileges used by the process that should be monitored (e.g. if the process to monitor was run as root `quickhpc` has to be run as root as well). When running `quickhpc` the required arguments are the PID of the process to monitor and the list of events to be monitored. Optionally it is possible to specify the maximum number of samples and the interval in microseconds between two samples.

The library used for probing HPCs is `PAPI` (Performance Application Programming Interface) [22]. The main reason why `quickhpc` uses `PAPI` is because of its resolution. After a thorough optimization `quickhpc` reaches a maximum resolution (i.e. the time between two samples) of 3 microseconds, more than 30000 times faster than `perf-stat`.

Events monitored by PAPI starts with the prefix `PAPI_`, for example `PAPI_L3_TCA` counts the total number of L3 cache accesses thus far; for periodic sampling it is sufficient to reset the counter of the event between two samples.

It is worth noting that the resolution of `quickhpc` is not fixed but is influenced by the workload on the system, the scheduling policy, the process monitored and so on. Also the number of collected samples, for the same process, may vary each time since it is not possible to start the monitored process and `quickhpc` at exactly the same instant; it all depends on the scheduling policy set in the operating system.

## 2.6. Anomaly detection

Anomaly detection is used to find outliers, or anomalies, in an unlabeled dataset. Some examples of real world problems, where anomaly detection plays an important role, are detection of faulty products in factories and detection of fraudulent transactions.

The assumption is that there exists a set of features, for each instance, or sample, in the dataset, that can let us determine whether the instance belongs to a specific model (e.g. "legitimate transactions") or not. Let us indicate with $x_j^{(i)}$ the j-th feature of the i-th instance in the dataset.

The aim is to retrieve a good number of samples considered "good" and find a probabilistic model that fits them. A usual assumption is that each feature $x_j$ fits a Gaussian distribution with mean and variance relative to that feature's values across all samples.

Therefore finding a model for feature $x_j$ means finding $\mu_j$ and $\sigma_j^2$ such that $x_j \sim \mathcal{N}(\mu_j, \sigma_j^2)$. Once these values are found the model can be tested by computing the distribution's density function for a new sample (i.e. probability that the given value $x$ belongs to a Gaussian distribution with the given mean and variance):

$$p(x_j; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(x_j-\mu)^2}{2\sigma^2})$$

The value returned by this function has to be compared to a threshold which can be determined, in turn, by testing the model on a dataset that contains known anomalies. This allows to find a threshold that clearly separates the anomalies from the normal samples, very similarly the purpose of a classifier in the domain of supervised learning explained in the next section. For each new sample this probability is computed for each feature. The total probability is computed as follows:

$$p(x^{(i)}) = p(x_1^{(i)}; \mu_1, \sigma_1^2)p(x_2^{(i)}; \mu_2, \sigma_2^2)...p(x_n^{(i)}; \mu_n, \sigma_n^2)$$

In our experiments we tried to fit a model for each kind of spy process implementation and considered all other (benign) processes as anomalies. The reason for not acting in the opposite way is that it is usually impossible to fit a model for all kinds of processes running on a system.

## 2.7. Supervised Learning and Neural Networks

The purpose of supervised learning is to construct models (classifiers) that are able to make predictions based on labeled data that were previously collected. Unlike unsupervised learning (where the purpose is to find patterns in non-labeled data) a datum, or sample, fed to a classifier, for the training phase, contains a vector of values named

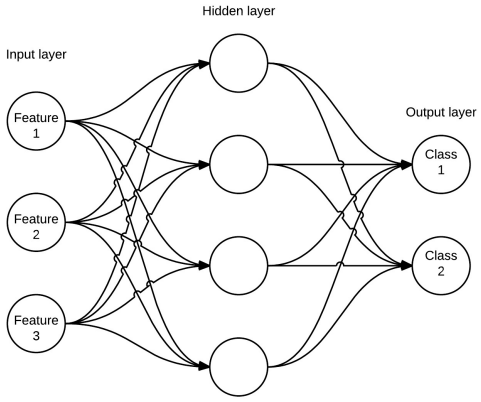


Figure 4. Representation of a simple neural network

features (or independent variables) and a label whose value is a function of them (dependent variable).

The classifier is then trained by using a relatively large number of samples, aggregated in what is called the training set. Upon completion of each training phase a cross-validation and a test set, consisting of data not present in the training set, can be used to assess the effectiveness of the classifier.

Once the training phase is complete it is possible to feed the classifier with a single vector of features, omitting a label, and it should be able to predict which class of entities the vector of features belongs to (the confidence of the prediction being dependent on a wide range of parameters). The hardest task in supervised learning is to find features that well characterize a certain class.

Although the principles behind supervised learning are similar to the ones behind anomaly detection there are a few key differences. In anomaly detection the classes are naturally skewed since the number of positives, i.e. anomalies, and negatives differ by several orders of magnitude (e.g. 1000 normal samples and 10 anomalies) where in supervised learning the more balance, between samples of different classes, the better; furthermore an anomaly detection mechanism does not distinguish between two or more classes but is only able to tell whether a sample belongs to the main class or not. In this paper we explored both options, taking into account their advantages and disadvantages.

Neural networks are intended to represent a set of classifiers inspired by how neurons collaborate in a brain to accomplish some tasks, hence the name. A commonly used model is the feedforward network.

As shown in Figure 4 such a network is formed by multiple layers. Each layer contains a certain number of neurons (or units), that hold a numerical value, called weight, and neurons of adjacent layers are connected to each other.

The vector of features is distributed among the neurons in the input layer and, after executing a feedforward propagation, the neurons in the output layer contain a vector of values whose maximum determines the class, i.e. its index in the vector, that the features supposedly belong to.

To train a neural network the feedforward propagation is followed by a backpropagation [29] step. In this step an error on the prediction is computed. Such error is then utilized, through the gradient descent algorithm, to adjust the weights of the neurons in the hidden layers to improve the accuracy of future predictions.

A common problem with supervised learning is overfitting. That is, the neural network precisely fits the training set but performs poorly on new, unlabeled samples. Such problem is partially solved by applying a technique named *regularization* [30] during the training phase.

The metric we used to assess how well both the neural network and the anomaly detection system performed is their F-score [28]. This metric is more reliable than merely measuring accuracy (i.e. right predictions over all predictions) since it is not influenced by datasets where some classes contain a larger number of samples than others, called *skewed classes*.

Thanks to neural networks we are able to devise a more sophisticated mechanism for detecting a spy process, compared to correlation and anomaly detection, that decreases the chances of incurring in false positives and serves as an initial attempt to detect spy processes that employ strategies to avoid being uncovered.

## 3. Detecting a spy process

In this section we present three methods for detecting spy processes that exploit the FLUSH+RELOAD technique to perform cache-based side-channel attacks.

All detection methods can successfully detect a spy before the attack is complete, therefore allowing to take appropriate countermeasures in time to prevent a leakage.

The first method is based on finding a correlation between the victim and the spy by analyzing the data collected by `quickhpc`. The intuition is that in all the attacks we analyzed, both the spy and the victim process behave approximately the same way: they execute a loop in which the same operations are performed. In our experiments we empirically established that a good indicator of correlation is the number of total L3 cache accesses over time.

In our experiments we tried to trigger false positives by simulating realistic workloads. Different kinds of operations, with different degrees of concurrency, are generated to stress an instance of the Apache web server while serving three different types of content: a small HTML file, a 1 MB JPG image and the result of a PHP script that calls functions that print information about the system. This choice was dictated by the fact that this kind of attacks mainly targets servers.

Although our experiments did not show any false positive we devised two more methods, based on machine learning techniques, that operate in a more fine-grained manner and therefore can be used to detect a spy with more confidence.

The second method makes use of an interesting machine learning technique: neural networks. Although computationally more expensive to train, they usually give better results than other supervised learning techniques [23] and

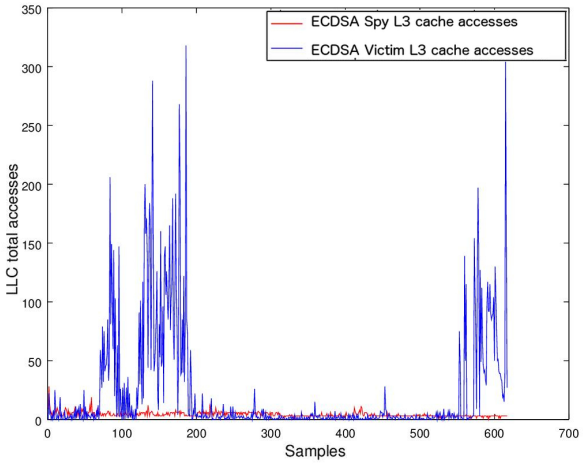


Figure 5. Total L3 cache accesses of spy and victim of the attack to ECDSA. The similarities are visible between samples 200 and 550 when the Montgomery ladder loop is executed.

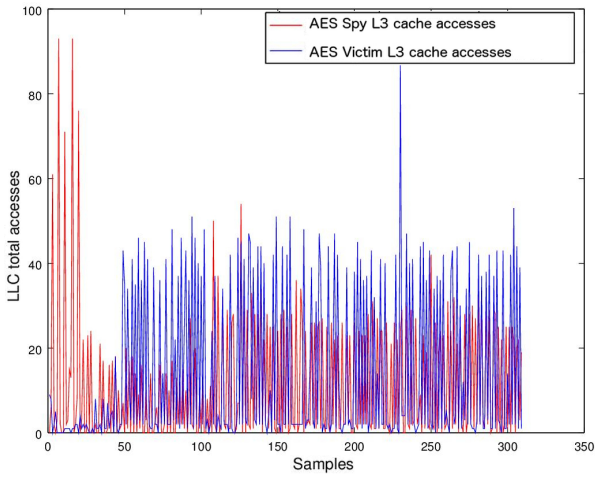


Figure 6. Total L3 cache accesses of the spy and the victim of the attack to AES.

do not require the data to be preprocessed (e.g. apply feature scaling and mean normalization).

Even though there exist many other supervised learning techniques, the good results yielded by our neural network convinced us to explore an option based on unsupervised learning instead of iterating over multiple supervised learning models.

The third method is based on treating the data samples coming from the spy process as ”normal” and data samples coming from any other process as anomalies or outliers. For such purpose we employed anomaly detection, an unsupervised learning technique. We were then able to determine whether a process is benign if it is recognized as an anomaly.

The downside of using anomaly detection or supervised learning is that there has to exist data that profiles a sample spy process, similarly to anti-virus applications that require a sample of the malware to be able to recognize it.

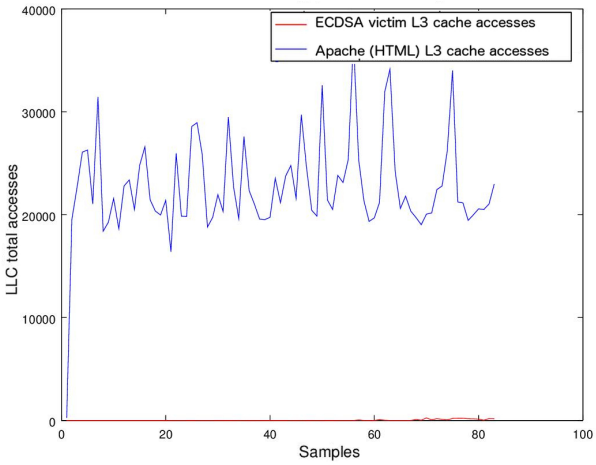


Figure 7. Total L3 cache accesses of the Apache webserver serving a 211 byte HTML file 1000 times with 100 concurrent clients and the victim of the attack to ECDSA.

### 3.1. Correlation-based approach

The intuition is that both processes spend most of their time in a loop where there is a regular access to potentially cached data. Without loss of generality, with regard to the other attacks, let us analyze the Montgomery ladder implementation, in the point scalar multiplication function `ec_GF2m_montgomery_point_multiply` of OpenSSL exploited in the second version of Yarom's FLUSH+RELOAD implementation [7].

The function contains a for loop, shown in Figure 2, in which the ephemeral key (the scalar used in the multiplication) is scanned bit by bit. Depending on the value of this bit a different conditional branch is evaluated at each iteration where the same two functions (`gf2m_Madd` and `gf2m_Mdouble`) are called with the arguments in a different order. This constant-time implementation should ensure that no useful information leaks through time while executing such function.

The spy, though, in this case is able to time the access to the first function in the first branch and to the second function in the second branch. This allows to guess, with high probability, which branch was chosen and, therefore, the value of the last bit of the word.

Since the loop is executed a large number of times it is fair to presume that its instructions will be loaded in the CPU's cache. In fact, as shown in Figure 5, between samples 200 and 550, the number of L3 cache accesses over time, while executing the main loop of the Montgomery ladder, is a value oscillating between approximately 0 and 25. The same kind of behavior can be observed, with regard to the AES spy and victim processes, in Figure 6 where the L3

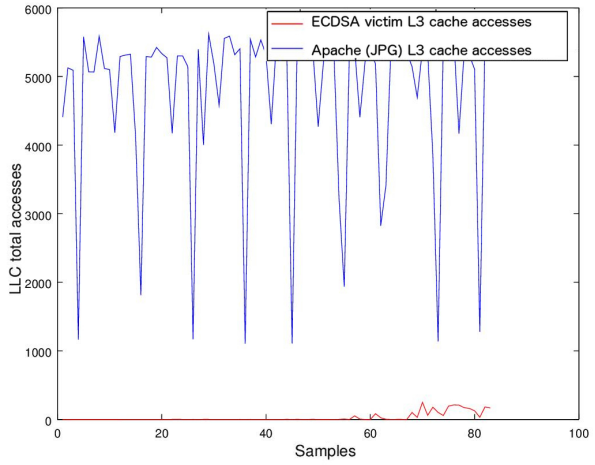


Figure 8. Total L3 cache accesses of the Apache webserver serving a 1 MB JPG file 1000 times with 100 concurrent clients and the victim of the attack to ECDSA.

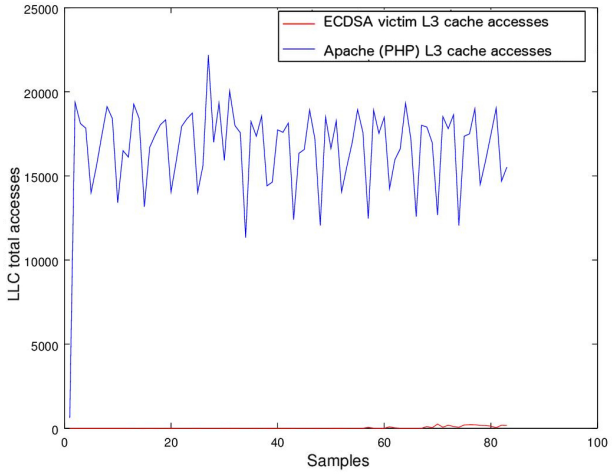


Figure 9. Total L3 cache accesses of the Apache webserver serving the output of a PHP script calling `php_info` 1000 times with 100 concurrent clients and the victim of the attack to ECDSA.

cache accesses over time for the two processes, after sample 50, almost overlap.

It is important to note that even though a piece of data is not present in the CPU's cache, each access to it will be registered as an access to the LLC. The MMU (Memory Management Unit) will then take care of triggering a cache miss, stall the process and eventually load the necessary data from the main memory into the cache and resume its execution.

Not surprisingly the spy process follows a similar pattern. The core of the computation lies in a loop where the process continuously flushes and reloads specific addresses from and into the cache. In this case the addresses of interest are the ones of the functions `gf2m_Madd` and `gf2m_Mdouble`.

This regularity is a requirement for the attack to work. In fact, as mentioned in the previous section, the spy has to synchronize with the victim to maximize the chances of success.

**Algorithm 7** Detect a spy process through correlation

```
1:  procedure DETECT-CORR(victim_PID, process_PID)
2:      s_1 = [ ]                                  ▷ s = samples
3:      s_2 = [ ]
4:      pipe(quickhpc(victim_PID), s_1)
5:      pipe(quickhpc(process_PID), s_2)
6:      while True do
7:          if correlation(s_1, s_2) > threshold then
8:              process_PID is likely a spy!
9:              break
10:         end if
11:     end while
12: end procedure
```

Such behavior can be exploited by monitoring both the victim and the spy at the same time and check how similar the number of LLC accesses over time is, as shown in Algorithm 7. In a real scenario it is often impossible to know when an attack of this sort is in progress therefore it is mandatory to continuously monitor a potential victim process and, separately, each new process spawned by the system.

The variant of the attack by Irazoqui et al., targeting AES, uses a similar mechanism to determine the key used in the last round of an encryption. The substantial difference is that their implementation uses a client-server architecture to trigger the encryptions and repeats the operation thousands of times.

Thanks to the high number of iterations the spy is, in this case, able to retrieve 100% of the bits of the last round's key. A major drawback of this approach is that it is easy to detect even by using lower resolution tools like perf-stat. In fact, given the 100 ms minimum resolution of perf-stat and assuming an execution time of 5 seconds, we are able to collect 50 samples, sufficient to determine whether there is correlation or not.

Since only a few milliseconds are sufficient to determine, with high accuracy, whether there exists a correlation between two processes, the monitoring phase does not affect their overall performance. Furthermore, while performing the experiments, it was noticed that the overhead caused by the monitoring tool is negligible.

Figures 5 and 6 show how visible this correlation is in both variants while Figures 7, 8 and 9 show how the number of L3 accesses over time differs significantly between the tested benign processes and the victim.

## 3.2. Based on anomaly detection

The reason why methods based on machine learning techniques might be needed is the potential presence of

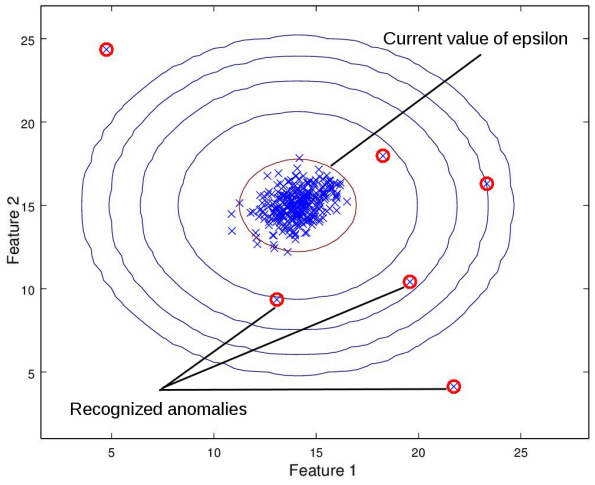


Figure 10. In this example different circles representing distinct values of epsilon, the threshold for the density estimation function, visually show how anomalies are flagged according to the value picked.

false positives (that is, there might exist processes that are benign but behave in a similar manner to a spy and would erroneously be flagged as malicious) and of a more sophisticated spy process which might find a way to escape the detection system based on correlation by creating noise, on purpose, to confuse the detection mechanism (such scenario is discussed further in Section 5). Utilizing machine learning techniques allows to profile this behavior as well, increasing the confidence of the detection.

In both methods based on machine learning we chose the following events monitored by `quickhpc` as features: total instructions, total CPU cycles, L2 cache hits, L3 cache misses, L3 cache total accesses. These events were empirically selected after various trials by analyzing the F-score reached for each feature set.

By using anomaly detection we can treat the data samples coming from the spy as normal and the data samples coming from other processes as anomalies. Like in supervised learning there is a "training" phase where the system is given some samples from the spy process. The training consists of three phases that are repeated until an optimal threshold $\epsilon$ is found:

1) Find $\mu_j$ and $\sigma_j^2$ for each feature $j$.
2) Compute $p(x)$ for each sample $x$ and find a value $\epsilon$ such that if $x$ is an anomaly $p(x) < \epsilon$.
3) Test $p(x)$ on a dataset that contains anomalies and verify that such anomalies are recognized.

Figure 10 shows how a small $\epsilon$ increases chances of recognizing a normal sample as an anomaly while a big value yields the opposite result. The optimal value of epsilon is chosen according to the F-score reached on the cross-validation set at each iteration.

Once this phase is complete the system can be used on new data.

### 3.3. Based on supervised learning

Another way of detecting a spy process, by analyzing its behavior at runtime, is to profile it in order to construct some kind of "signature" that can be used to identify it with a certain confidence, similarly to what anti-virus software does with static signatures.

In the context of supervised learning the profiling phase translates into a training phase for the classifier (in this case a neural network). The raw data collected by `quickhpc` is first processed by a set of scripts, merged together in a single dataset and fed to the neural network.

The output neurons represent the two classes of interest: malicious process and benign process. The victim process is labeled as benign in the training set. The presence of samples from the victim is useful to make the network differentiate between two processes that have a very similar behavior (as shown by their correlation) but belong to different classes.

## 4. Experiments and results

All our experiments were performed on an HP Z400 workstation with a Intel Xeon W3670 CPU, operating at a manually fixed clock of 3.2 Ghz, and 20 GB of RAM. The operating system used was Ubuntu 14.04 LTS with kernel Linux 3.13.0-46-generic.

**Algorithm 8** Compute correlation coefficient between two datasets

**procedure** CORRELATION($data_1$, $data_2$)
  $samples = min(data_1.size, data_2.size)$
  $diff = data_1 - data_2$
  $cv = cov(diff)$
  $confidence = samples * (1/cv)$
  **return** $confidence$
**end procedure**

For each type of attack we performed 100 iterations where we monitored the spy, the victim and a benign process operating in different contexts. Each iteration is divided into the following phases:

1) Execution and monitoring of the victim process
2) Execution and monitoring of the spy process
3) Execution and monitoring of the benign process
4) Data analysis and prediction

All processes are launched at the same time. Once the spy successfully completes an attack all monitored processes are terminated and the analysis phase begins. During this phase we feed the data to three scripts: one that implements the correlation system, one that implements the anomaly detection system and one that implements the neural network. Each script reports the number of samples used, the confidence of the detection and the time it took to complete it.

TABLE 1. BENCHMARKS OF THE DETECTION METHOD BASED ON CORRELATION

| **Correlated processes (100 iterations)** | **Min confidence (samples)** | **Max confidence (samples)** |
|---|---|---|
| AES spy with AES victim | 0.094715 (42) | 5.4 (522) |
| ECDSA spy with ECDSA victim | 0.001565 (21) | 1.66 (744) |
| Apache (HTML file) with AES victim | 0.000002 (42) | 0.000008 (157) |
| Apache (JPG file) with AES victim | 0.000028 (42) | 0.000398 (862) |
| Apache (PHP file) with AES victim | 0.000004 (42) | 0.000163 (862) |
| Apache (HTML file) with ECDSA victim | 0.000001 (36) | 0.000008 (157) |
| Apache (JPG file) with ECDSA victim | 0.000007 (11) | 0.000566 (1422) |
| Apache (PHP file) with ECDSA victim | 0.000002 (29) | 0.000295 (1422) |
| Time to find correlation over 500 samples | 0.35 ms | |

TABLE 2. BENCHMARKS OF VARIOUS OPERATIONS

| | **Time** |
|---|---|
| ECDSA signature Montgomery Ladder loop (OpenSSL, curve sect571r1) | 2.8 ms (default compilation flags)<br>9.5 ms (with debug symbols enabled) |
| ECDSA signature Total time (OpenSSL, curve sect571r1) | 6 ms (signed 1 B file)<br>9 ms (signed 1 MB file) |
| ECDSA spy Minimum time needed | 2.8 ms (the time it takes to complete a single Montgomery ladder loop) |
| AES spy Minimum time needed | 5 s (same OS scenario) |
| Maximum quickhpc resolution | 3 $\mu s$ (measured with {clock_gettime()}) |

TABLE 3. BENCHMARKS OF THE DETECTION METHODS BASED ON MACHINE LEARNING TECHNIQUES

| **Method** | **Max F-score** | **Time for prediction (over 100 samples)** |
|---|---|---|
| Anomaly detection (AES) | 0.509091 | 0.2 ms |
| Anomaly detection (ECDSA) | 1.0 | 0.2 ms |
| Neural network (AES) | 0.932331 | 0.64 ms |
| Neural network (ECDSA) | 1.0 | 0.64 ms |

The correlation coefficient is computed as in Algorithm 8. The confidence that a correlation exists is given by the following formula:

$$confidence = samples * (1/variance).$$

Table 1 below gives a quantitative insight on how such value changes according to the type of attack we try to detect. With respect to the spy process used while attacking AES the range of confidence varies from a minimum of 0.095 to a maximum of 5.4 but when attacking ECDSA the minimum and maximum confidence values drop to around 0.002 and 1.66 respectively. It is clear that this value is influenced by the number of samples `quickhpc` was able to process and the higher the number of samples the higher the chance of getting a good level of confidence (as shown in Figure 11).

For what concerns the benign processes the range decreases significantly with a minimum of $10^{-6}$ and a maximum of $5.66 * 10^{-4}$ which ensure the absence of false positives since the latter value is roughly one order of magnitude lower than the minimum confidence given by any spy process.

The execution time for both the AES and ECDSA victim processes is reported below in Table 2 where the fastest operation is the execution of the Montgomery Ladder loop, shown in Figure 2, that takes a maximum of 2.8 ms.

On our system the time to execute Algorithm 8 over a dataset of 500 samples is 0.35 ms on average. Considering the fastest implementation of the attack has a minimum execution time of 2.8 ms (i.e. the duration of the Montgomery ladder loop in OpenSSL) there are still 2.45 ms that can be used to take appropriate countermeasures.

The performance of the neural network is a little worse but good enough for our purposes. Within 0.64 milliseconds the network completes the feedforward propagation over 100 samples and returns the predicted class (spy or not). In this case the confidence is measured as follows:

$$confidence = predictions_{spy}/predictions_{total}$$

On the other hand the anomaly detection system, for a prediction over 100 samples, only takes 0.2 ms on average making it the fastest one. Unfortunately it is also the one that suffers the most from noisy data making it perform poorly on certain datasets as shown in the next section. Even in this case the confidence is computed with the aforementioned formula.

F-scores for both the anomaly detection system and the neural network are reported in Table 3 together with the

time it takes to perform a prediction (i.e. to classify) over 100 samples.

### 4.1. Overhead

In our methods we consider the case where `quickhpc` can be arbitrarily attached to any running process and collect samples for as long as it is needed (the minimum number of samples needed for detection varies according to the type of spy process as shown in the next sections).

Such approach is applicable only if the overhead caused by `quickhpc` is negligible. To determine the overhead caused by `quickhpc` we performed 1000 OpenSSL ECDSA signatures with curve `sect571r1` while the victim process was being monitored and while it was not. The average execution time was 6 ms in both cases.

To make sure the parallelism offered by a multi-core architecture was not responsible for such low overhead (i.e. because the crypto process and `quickhpc` were being executed on different cores) we "pinned" all of the processes to a single core by using the utility *taskset* [32].

### 4.2. Detecting AES spy process

As described in [9] finding the last round's key in an AES encryption, by using a variant of the FLUSH+RELOAD technique, takes a varying amount of time in the order of seconds to minutes.

The execution time depends on the scenario in which the attack is carried out. If both the spy and victim processes are being executed within the same operating system the attacks takes a few seconds; 5 on average on our test system while still being able to recover all the bytes of the key. If the spy and the victim are on separate virtual machines, although share the same CPU, as it often happens with virtualization services like Digital Ocean [27] or Amazon EC2 [31], the execution takes approximately one minute.

Given how long the spy needs to execute, for a successful attack, quickhpc is able to collect a very large number of samples in a short amount of time. In our experiment we let the spy and the victim run for 50 ms, 100 times less than the average time needed to complete the attack.

Figure 11 shows the confidence of the detection according to the number of samples collected. As expected the more the samples the higher the confidence with the minimum (0.095) reached with 42 samples and the maximum (5.4) reached with 522 samples.

The minimum confidence with regard to the benign processes reached a maximum value of 0.0005, two orders of magnitude less than the minimum confidence with regard to the spy, effectively eliminating the chance of incurring in false positives.

The methods based on machine learning performed very differently in this case. The anomaly detection system performed poorly with a maximum F-score of 0.51 while the neural network reached instead an F-score of 0.93.

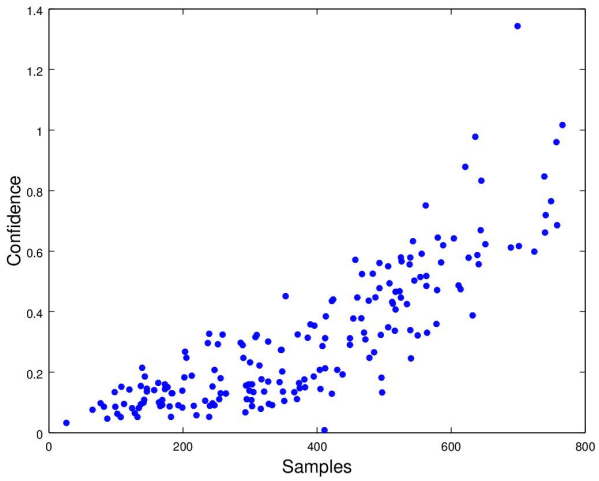


Figure 11. Relationship between the number of samples collected during an attack to AES and the confidence of the prediction based on correlation. Even though the relationship is not linear (since the confidence is influenced by noise caused by other processes, scheduling policies etc.) the general trend is that the higher the number of samples the higher the confidence.

### 4.3. Detecting ECDSA spy process

A complete signature of a 1 byte file, using ECDSA with OpenSSL, takes 6 ms on average while using a 1 Megabyte file increases this time by 3 ms for a total of 9 ms.

The main loop used in the Montgomery ladder implementation lasts 2.8 ms on average which means that, since both detection methods take approximately 0.2 to 0.64 ms, there are around 2-2.5 ms left to take countermeasures, assuming a successful attack is complete once all the bits of the ephemeral key have been scanned.

Considering a resolution of 10 $\mu s$, for `quickhpc`, we could obtain, in 2.8 ms, roughly 280 samples. The resolution varies according to how the system is performing (i.e. how many processes are running, how the scheduler acts with regard to `quickhpc` and the monitored process etc.) so the number of samples obtained, and thus the sampling resolution, might be more or less than this theoretical value.

The minimum confidence reached by determining a correlation between the victim and the actual spy was of approximately 0.0016 with 21 samples while the maximum was 1.66 with 744 samples.

Even in this case the maximum confidence for the correlation between a benign process and the victim was almost one order of magnitude less than the minimum confidence for the correlation between the spy and the victim.

Both machine learning methods, though, performed well with an F-score of 1. Since the time to perform a prediction over 100 samples does not change according to the samples themselves even in this case it took 0.2 and 0.64 ms on average respectively for the anomaly detection system and the neural network.

TABLE 4. CONFIDENCE VALUES AND EXECUTION TIME FOR THE THREE VARIANTS OF THE MODIFIED VERSION OF THE AES SPY PROCESS. EACH VARIANT SETS A DIFFERENT VALUE FOR THE MAXIMUM NUMBER OF ADDRESSES THAT ARE PROBED FOR EACH ITERATION.

| Number of random accesses | Min confidence | Max confidence | Max time to complete an attack |
|---|---|---|---|
| 10 | 0.063545 | 3.197853 | 19s |
| 100 | 0.159697 | 1.229513 | 43s |
| 1000 | 0.003177 | 0.355517 | 96s |

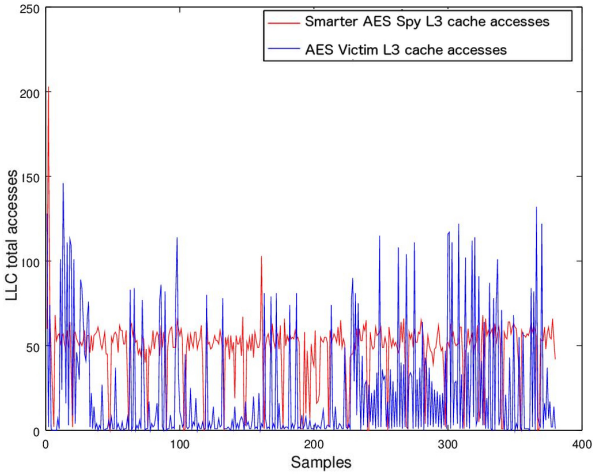


Figure 12. Relationship between the total LLC accesses of the AES victim process and the modified version of the spy process.

## 5. A smarter spy process

The purpose for building a more sophisticated version of a spy process is to evade one or more of the detection systems presented in the previous sections.

We were able to decrease the confidence range given by the first system, based on correlation, by slightly changing the behavior of the spy so that it would take more time to complete an attack but act in a more clever way. We chose to modify the spy for AES by Irazoqui et al. [9] because of its already long execution time (i.e. minimum 17s on our system).

Since the correlation is established only by the total number of cache accesses, the modified spy, similarly to what happens in the actual attack, can start accessing a random number of addresses generating, therefore, a random number of cache hits or misses: accesses nonetheless. Although these random accesses, performed for each iteration of the main loop of the spy, cause the total execution time to increase, the success of the attack is in no way influenced. Figure 12 shows the relationship between the total cache accesses of the AES victim and the AES modified spy; a very different pattern than the one previously seen in Figure 6 with the original spy process.

We modified the spy to access up to 10, 100 and 1000 addresses for each iteration of the main loop. In all cases the key was correctly retrieved, proving the attack can still be completed, even though the execution time increased dramatically, up to 96 s, in the last case. On the other hand the confidence range noticeably decreased. The minimum value went from 0.095, for the original spy, to 0.003 for the modified version while the maximum dropped from 5.4 to 0.35. This proves that it is possible to partially circumvent the detection system based on correlation while still being able to successfully complete an attack.

Table 4 shows how the confidence range depends on the number of random addresses used. The data have been collected over 100 attacks for each number of random addresses.

We experienced the opposite trend when trying to catch such process by using the neural network and anomaly detection system. In the first case the maximum F-score was 0.98 while in the second case the value dropped to 0.79, similarly to the unmodified AES spy process. The new behavior clearly makes the process stand out more, rendering the detection even easier when using techniques based on machine learning.

## 6. Discussion

Our results show that it is possible to catch a process that uses the `FLUSH+RELOAD` technique before the attack can be successfully completed. The fact that our detection system can run as a process in userspace makes it convenient to use both on a same-OS scenario and on virtual machines.

In the second scenario the choice would be to either integrate the system into the hypervisor or preinstall the software on any new virtual machine, as it happens with VMWare tools.

In a same-OS scenario the time left between the completion of the detection and the completion of the attack, in the case of the fastest spy where there are 2.6 to 2.2 ms left, allows for a variety of countermeasures, the simplest being killing the suspicious process and prevent further access to any file or socket opened by it. In case of a cross-VM attack it would be enough, for the hypervisor, to suspend the virtual machine where the spy is running and relocate the one where the victim is running since co-location is the first requirement for this kind of attacks to work.

The creation of a *smarter* spy process proved that the detection based on correlation can be partially circumvented opening the doors to further research on how to implement a more advanced variant of the aforementioned attacks. Deceiving the other detection systems, based on machine learning techniques, proved to be a harder task, although the assumption that there exist training data might not always be correct when encountering new variants that work in unexpected ways (which often happens with antivirus software).

The low footprint generated by our system and its ability to run as yet another userspace process, together with the fact that most systems are not regularly patched against such attacks, make it a good tool for cloud services providers. We think the best way to employ our detection system would be to integrate it with the process responsible for the creation of all other processes (e.g. `init` in Unix-like operating systems). It would be enough to attach `quickhpc` to each new process, monitor them for a predefined amount of time, run one (or all) of the detection algorithms on the collected data and decide whether to terminate the process or simply detach `quickhpc` and let the process run.

## 7. Conclusion and future work

In this paper we introduce three methods to detect a spy process that is performing a cache-based side-channel attack based on techniques like `FLUSH+RELOAD` (in general any technique where the attacker is required to periodically access the CPU shared cache).

While each of the methods has its own strengths and weaknesses we proved that it is definitely feasible to detect and prevent an attack, from being completed, in a relatively short time. Furthermore we did so without altering any of the components of the system (e.g. the kernel) and without decreasing its performances, simply by running our detection system as a user space process.

We are confident that such system might be easily integrated in a physical or virtual cloud environment (such as DigitalOcean or Amazon EC2) either as a separate process (similarly to an anti-virus) or as a plugin for the hypervisor.

On the other hand we also demonstrated how, with just some tweaks, it is possible to deceive one (the simplest) of the detection methods. This, we hope, will fuel more research on increasingly "smarter" detection systems and, consequently, attacks.

## Acknowledgments

We would like to thank the authors of [7], [8] and [9] for sharing, and assisting us with, the source code of their projects.

## References

[1] Tian Tian, Chiu-Pi Shih. *Software Techniques for Shared-Cache Multi-Core Systems*. Intel Developer Zone. 2012.

[2] Andrea Arcangeli, Izik Eidus, Chris Wright. *Increasing memory density by using KSM*. Red Hat, Inc. 2009.

[3] Ganesh Venkitachalam, Michael Cohen (VMWare, Inc.). *Transparent page sharing on commodity operating systems*. Patent US7500048 B1. 2009.

[4] Ristenpart, Thomas, et al. *Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds.*. Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009.

[5] Tsunoo, Yukiyasu, et al. *Cryptanalysis of DES implemented on computers with cache.* Cryptographic Hardware and Embedded Systems-CHES. 2003.

[6] Osvik, Dag Arne, Adi Shamir, and Eran Tromer. *Cache attacks and countermeasures: the case of AES.* Topics in CryptologyCT-RSA. 2006.

[7] Yarom, Yuval, and Katrina E. Falkner. *Flush+ Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack.* IACR Cryptology ePrint Archive. 2013.

[8] Yarom, Yuval, and Naomi Benger. *Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack.* IACR Cryptology ePrint Archive. 2014.

[9] Irazoqui, Gorka, et al. *Wait a minute! A fast, Cross-VM attack on AES.* Research in Attacks, Intrusions and Defenses. Springer International Publishing. 2014. 299-319.

[10] Glmezoglu Berk, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. *A Faster and More Realistic Flush+ Reload Attack on AES.*. 2015.

[11] Sprunt, Brinkley. *The basics of performance-monitoring hardware.* IEEE Micro 4 pp. 64-71. 2002.

[12] Jonathan Corbet. *KSM tries again*. LWN. 2009.

[13] Fei Guo. *Understanding Memory Resource Management in VMware vSphere 5.0*. VMWare Performance Study. 2011.

[14] Anonymous. *Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing (2080735)* . VMWare knowledge base. 2015.

[15] Salvador Palanca, Stephen A. Fischer, Subramaniam Maiyuran (Intel Corp.). *CLFLUSH micro-architectural implementation method and system*. Patent US6546462 B1. 2003.

[16] Messerges, Thomas S., Ezzy A. Dabbish, and Robert H. Sloan. *Power analysis attacks of modular exponentiation in smartcards*. Cryptographic Hardware and Embedded Systems. Springer Berlin Heidelberg, 1999.

[17] Brown, D. *SEC 2: Recommended Elliptic Curve Domain Parameters*. 2010.

[18] Joye M., Yen S. M. *The Montgomery powering ladder*. In Cryptographic Hardware and Embedded Systems-CHES (pp. 291-302). 2002.

[19] Uht, Augustus K., Vijay Sindagi, and Kelley Hall. *Disjoint eager execution: An optimal form of speculative execution*. Proceedings of the 28th annual international symposium on Microarchitecture. IEEE Computer Society Press, 1995.

[20] Ammons, Glenn, Thomas Ball, and James R. Larus. *Exploiting hardware performance counters with flow and context sensitive profiling*. ACM Sigplan Notices 32 pp. 85-96. 1997.

[21] de Melo, Arnaldo Carvalho. *The new linux perf ools.* In Slides from Linux Kongress. 2010.

[22] Mucci, Philip J., Shirley Browne, Christine Deane, and George Ho. *PAPI: A portable interface to hardware performance counters.* In Proceedings of the Department of Defense HPCMP Users Group Conference, pp. 7-10. 1999.

[23] Caruana, Rich, and Alexandru Niculescu-Mizil. *An empirical comparison of supervised learning algorithms.* In Proceedings of the 23rd international conference on Machine learning, pp. 161-168. ACM, 2006.

[24] Uhsadel L., Georges A., Verbauwhede I.. *Exploiting Hardware Performance Counters*. 5th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). 2008.

[25] Adrian T. et al. *Unsupervised Anomaly-based Malware Detection using Hardware Features.* 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID). 2014.

[26] Nishad Herath, Anders Fogh. *These are Not Your Grand Daddy's CPU Performance Counters*. Black Hat USA. 2015.

[27] Digital Ocean. https://www.digitalocean.com/. Last retrieved: August 2015.

[28] Van Rijsbergen, C. J. *Information Retrieval (2nd ed.)*. Butterworth. 1979.

[29] Werbos, P.J. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.*. 1975.

[30] Christopher M. Bishop. *Pattern Recognition and Machine Learning, pp. 256-269*. 2007.

[31] Amazon EC2. https://aws.amazon.com/ec2/. Last retrieved: August 2015.

[32] Taskset. http://linuxcommand.org/man_pages/taskset1. Last retrieved: August 2015.

[33] QuickHPC. https://github.com/lambdacomplete/quickhpc. Last retrieved: August 2015.