**manuscript No.**
(will be inserted by the editor)

# Spy Cartel: Parallelizing Evict+Time Based Cache Attacks on Last Level Caches

**Himanshi Jain** · **D. Anthony Balaraju** · **Chester Rebeiro**

**Abstract** A powerful cache timing attack can not only determine the secret key of a cryptographic cipher accurately but also do so quickly. Cache timing attacks that utilize the shared L1 cache memory are known to have these two characteristics. On the other hand, attacks using the shared Last Level Cache (LLC) memory are not always successful in obtaining the secret key, and they take considerably longer than an L1 cache attack.
This paper leverages the fact that all LLC attacks run on multi-core CPUs, facilitating the attack programs to be parallelized. We show how parallelization can be used to reduce the runtime and improve the attack's success making it on a par with L1 cache attacks. We then propose a new methodology for LLC cache attacks, by which, an attacker can maximize the attack success for a given timeframe. The only additional requirement is learning about the target system's runtime behavior, which is done offline. We validate all our claims on a 4-core and a 10-core CPU.

**Keywords** Cache timing attacks, Last Level Cache memories, Evict+Time, Multi-core CPUs

## 1 Introduction

Cache timing attacks are a powerful form of cryptanalysis, where an attacker exploits the fact that the cache memory is shared in a system. The attacker runs a program, called the *spy*, which contends with a *victim* program for the shared

This work is supported by Information Security Education and Awareness Program (ISEA), Ministry of Electronics and Information Technology (MeitY), India and DST-FIST Grant Program 2016, from Department of Science and Technology, India. We also thank Mr. Raghavandra Patil from C-DAC Bangalore for his timely inputs.

Himanshi Jain, Indian Institute of Technology Madras, INDIA
E-mail: himanshi@cse.iitm.ac.in
D. Anthony Balaraju, Indian Space Research Organization, INDIA
E-mail: balaraju17@gmail.com
Chester Rebeiro, Indian Institute of Technology Madras, INDIA
E-mail: chester@iitm.ac.in

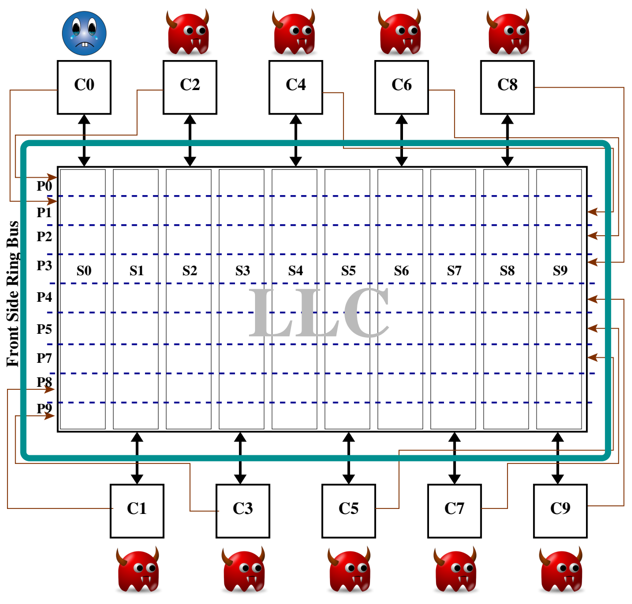


Fig. 1: A 10-core CPU with cores C0, C1, . . . C9, connected with a front-side ring bus and having a shared LLC with slices S0, S1, . . . , S9. Spy programs in LLC attacks can thus be parallelized to leverage the shared LLC. Parallelization can partition the LLC (P0, P1, . . . , P9) to reduce attack time or can be configured to increase attack success.

cache memory. This contention affects the number of cache misses, which is manifested in the execution time of both programs. Attackers have been able to monitor the execution time and glean sensitive information from the victim. Cache timing attacks have been known for about two decades. The first set of attacks used the shared L1 data or instruction cache [2, 3, 9, 11, 13, 16, 20, 23, 24, 30]. These attacks make a strong assumption that the spy executes in the same CPU core as the victim; albeit in a different hardware thread. Recent attacks relaxed this assumption, by targeting the last level cache (LLC) memory [4, 5, 17, 19, 22, 34]. In an LLC attack, the spy and victim can reside on different cores in a multi-core CPU and contention is for the shared cache lines in the LLC. This is a big threat especially to multi-tenant cloud computing service providers, who permit different users to simultaneously share different cores of the same CPU. Several forms of LLC attacks have been developed in the last few years such as [5, 17, 22, 34]. Many of these attacks are much more complex than the L1 cache attacks due to multiple reasons. First, the entire LLC is split into slices, and the mapping from physical address to slice, is based on a machine dependent undisclosed hash function. Second, the LLC is considerably larger than the L1 cache. The *interesting cache sets*, which hold victim's sensitive information, would be a small fraction of the entire LLC. Due to these reasons, creating a spy that contends for the same cache lines as the victim, is considerably difficult. Further, unlike L1 cache memories, which are indexed by virtual addresses, the LLC is indexed by physical addresses, which are not visible to the attacker and change at every execution. Thus, the spy would need to locate the interesting cache sets, every time the victim is restarted. Due to these factors, an LLC attack is several times slower than an L1 cache attack. For example, from our experiments a typical L1 cache attack on an implementation of AES[1] takes 1.08 minutes on average, while the same attack using the LLC takes 19.08 minutes on a 10 core machine. The LLC is also considerably slower than the L1 cache. This may result in a decreased rate at which the spy can contend for the shared cache sets. This in turn reduces the impact on the execution time of the programs, resulting in lower attack success.

In this paper we are concerned with optimizing LLC attacks based on the Evict+Time technique [23]. We leverage one characteristic difference between the L1 cache memory and the LLC: while the L1 cache is private to a CPU core, the LLC is typically shared between all cores. This allows the attacker to parallelize the spy so as to increase the attack success and/or reduce the time required to mount the LLC attack. For example, in a 10 core CPU, the victim would execute in one core, while 9 other spies would execute in the other cores; one in each core as shown in Figure 1. The spy threads can be configured to parallelize the search for the victim's interesting cache sets, thereby reducing the attack time. Alternatively, the attack's success can be boosted by focusing all spies on the interesting cache sets, thereby increasing contention, which in turn increases the attack success.

In this paper we first demonstrate how a parallel spy can be used to boost the attack success and/or reduce the attack time. We then present a novel attack methodology, by which an attacker can choose a spy configuration apriori, so as to maximize attack success within a given timeframe. Along the way, we make several other novel contributions, which we list below.

[1] OpenSSL ver. 1.0.1f (https://www.openssl.org/)

- While all state-of-the-art works on LLC attacks [4, 5, 17, 19, 22, 34] have focused on different attack techniques, ours is the first work that looks at LLC attacks with parallel spy threads. This is especially useful in LLC attacks because of the multi-core architectures of modern CPUs.
- We study different search techniques in order to identify the interesting cache sets in the LLC. We demonstrate how each search strategy can affect the attack runtime and accuracy of finding the interesting cache sets.
- Parallelizing the spy, results in flooding the front side ring bus (ref. Figure 1). We investigate the impact of flooding the bus on the success of the attack.
- For our experiments we compare two CPUs: a small 4-core CPU and a larger 10-core CPU. This permits the study of the relationship between the number of CPU cores and the success of a cache timing attack. On both the test systems, the proposed attack always outperforms the naïve attack technique, sometimes by over 60%. We also show that the proposed attacks are better on CPUs with larger number of cores.

The structure of the paper is as follows: Section 2 gives the necessary background required for this work. Section 3 has the related work on cache timing attacks, particularly focusing on LLC attacks. Section 4 presents an Evict+Time based cache attack, which we consider in this paper. First the overview is presented and then the details are discussed. Section 5 and Section 6 present optimizations for the various phases of the attack using a parallel spy. In Section 7, we present a generic algorithm to find the best strategy to parallelize the spy, given a bound on the attack time. Section 8 has the results of our attack, while Section 9 concludes the paper.

## 2 Background

In this section, we give a brief introduction to last level cache memories and cache timing attacks. We then define a metric which we use to quantify the attack success of a cache timing attack.

### 2.1 Last Level Cache Memories

Most current server class processors have three levels of caches, level 1 (L1), level 2 (L2), and last level cache (LLC or L3). The L1 and L2 caches are private to each core whereas the LLC is shared across all the cores of a CPU. Cache memories in most Intel processors adhere to the *inclusiveness property*, where data present in the L1 and L2 caches is a strict subset of the data present in the LLC. This feature has been exploited in all last level cache attacks.

Cache memories are organized as *cache lines*, typically of 64 bytes. Cache lines are further organized as cache sets. In the L1 cache, data is mapped into one of these cache sets using the *set index* bits of the virtual address. In the LLC however, the physical address determines the cache set. Since physical addresses are not known to attackers with user level privileges, cache attacks on the LLC are considerably more difficult. Besides this, there are a number of subtle differences in the organization of an LLC compared to an L1 cache, which further increases the attack difficulty.

The LLC is split into slices, with each CPU core associated with a slice. All the slices are connected with a front side ring bus and are accessible from any of the cores (ref. Figure 1). Due to slicing, the mapping of an address to a cache set is more complicated at the LLC compared to the L1 cache. Intel processors use an undisclosed hash function to uniformly distribute addresses across the slices [19, 35]. These hash functions, which are machine dependent, map a given physical address to any one of the LLC slices.

### 2.2 Evict+Time Cache Timing Attacks

In this attack, the attacker runs a spy process simultaneously with the victim process. The spy is designed to share a cache with the victim process and continuously performs some memory accesses. In parallel it also invokes the victim process and monitors the execution time. If the spy's memory accesses happen to interfere with the victim's memory accesses due to the shared cache memory, then there is an increase in the execution time of the victim. These execution time changes are then analyzed to reveal secret information about the victim [6, 12, 23, 30].

### 2.3 Measuring the Attack Success

We use a metric similar to guessing entropy [18] to gauge the effectiveness of an attack. Let $n_1$ denote the worst case number of guesses it takes for an attacker to determine the secret key of a cipher without any side-channel information. Further, let $n_2$ denote the number of guesses it takes to determine the secret key, given the execution time side-channel. We compute the success of the attack as follows:

$$\%AttackSuccess = \frac{|n_1 - n_2|}{n_1} \times 100 \ . \tag{1}$$

This metric indicates how much better the side-channel attack is compared to a random guess of the secret key. If for instance, the number of possible key values is 256. Then $n_1$ is 256. Suppose, given the side-channel leakage, the number of guesses required (*i.e.* $n_2$) is 10, then success of the attack is 96.1%.

## 3 Related Work

In the last decade, there were several works that demonstrated vulnerabilities of cryptographic ciphers due to the L1 cache. The attacks were classified as time-driven [6, 8, 14, 21, 25–27, 29, 31], trace-driven [1, 7, 15, 23, 36], or access-driven [11, 20, 23, 24, 30]. Many of these attacks assumed that the attacker shared a CPU core with the victim in a hyperthreaded environment. The fact that hyperthreading was disabled in many newer CPUs, and time-sharing of CPU cores were not permitted, led researchers to look beyond the L1 cache and utilize the LLC to mount cache attacks. The first LLC attack was based on Flush+Reload. It was introduced by Yarom and Falkner in [34]. An efficient Cross-VM Flush+Reload cache attack was later developed by Irazoqui *et al.* [4]. Both these attacks have the constraint that memory pages need to be shared between the attacker and the victim. In order to prevent these attacks many cloud services providers disabled de-duplication of memory pages. Recent works, managed to design LLC attacks without needing de-duplication. In these attacks, the attacker targets a small fraction of the LLC sets that are likely to leak the victim's secret information. The complications however, is that the number of sets in the LLC is considerably large, which slows down cache probing and also affects success. Intel's undocumented hash function that obfuscates cache mappings in the LLC further hampers the attacks. In 2015, Maurice *et al.* [19] and Yarom *et al.* [35] demonstrated how the hash function can be reverse engineered thus determining the exact mapping between memory addresses and LLC slices. The technique in [19] requires knowledge of physical addresses, which is not always possible. Around the same time, Irazoqui, Eisenbarth, and Sunar showed that even without sharing of pages, LLC attacks can be developed by leveraging huge pages [5]. Liu *et al.* in [17], also used huge pages in their attacks to identify conflicting sets. The algorithms that were proposed, eliminated the need for physical addresses to reverse engineer the hash function.

Most of the LLC attacks demonstrated so far are time consuming and in many cases inefficient. The unknown addresses and the undisclosed mapping between physical addresses and slices increases the complexity of the attacks further. In this paper we present optimization techniques to improve LLC attacks. The optimizations can be used to increase attack success or reduce attack runtime. We describe how an attacker can choose among these multiple optimization techniques to derive the best attack.

## 4 Evict+Time based Last Level Cache Attack

In this section, we present an Evict+Time based cache attack on the last level cache. We divide the entire attack into three phases: learning, scout, and strike. While the learning phase is done offline, the scout and strike are online phases.

### 4.1 Attack Assumptions

We assume that the attacker has sufficient time to reverse engineer the mapping between physical addresses and cache slices. During the attack, we assume that the attacker co-resides with the victim on the same CPU; not necessarily the same core. There are several techniques to achieve such co-location like [28, 32, 33]. We also assume that the LLC is shared between all cores in the CPU and the attacker can accurately measure execution time.

The attacker runs a spy process, which accesses the same cache set as the victim, thus interfering with the victim's execution. We assume that the attacker can accurately measure the disturbances in the victim's execution time due to the interference.

## 4.2 Overview of the Attack

The attack comprises of three phases:

- **Learning Phase.** We reuse techniques from [17] to determine conflicting lines in an LLC set. This involves finding an eviction and a conflict array. An eviction array $\mathbb{E}_i$ contains memory addresses that fills exactly one slice of an LLC set $i$, while a conflict array $\mathbb{C}_i$ contains all the eviction arrays corresponding to all slices of the cache set $i$. Thus a conflict array for an LLC cache set in a 10 core CPU would comprise of 10 eviction arrays; one for each slice.
- **Scout Phase.** In this step, we search for the interesting LLC cache sets that the victim uses. If the victim's virtual address space is known, for example compile time allocated data, then a few clues about the mapped cache sets are obtained. Thus, search is restricted to a small subset of the LLC sets. On the other hand, for dynamically allocated data or when the victim's virtual address space is not known, the entire LLC needs to be probed to identify the interesting cache sets used by the victim. The latter is especially the case when the victim is coded in a Languages such as Java or JavaScript, where the virtual address space is not easily observable. Since the physical address mapping may change every time the victim is restarted, the scout phase will have to be rerun, every time the victim's physical address mapping changes.
- **Strike Phase.** Once the attacker has identified the victim's interesting cache sets, she monitors execution time and then uses statistical techniques to derive sensitive information from the victim.

It is most important to optimize the scout and strike phases, since they are executed online. In this section we elaborate each of these phases, while the following sections present techniques to optimize the scout and strike phases.

**Learning Phase.** When a physical address reaches the LLC, a hash function selects the LLC slice, while the set index bits in the physical address identifies the cache set within the slice. This mapping is machine dependent. In the learning phase, we follow the approach of [17] to identify a set of addresses that gets mapped to the same cache set in a slice. We call such a set as the *eviction array* for the cache set $i$ and denote it by $\mathbb{E}_i$ . For a CPU with $M$ cores (*i.e.* $M$ cache slices), there are $M$ such eviction arrays for each cache set. The union of these eviction arrays is called a *conflict array* for the set $i$, and denoted by $\mathbb{C}_i$.

**Algorithm 1:** Identifying an eviction array for one slice of a cache set

```
Input: $\mathbb{E}$ : A set of at-least $w+1$ addresses in the huge array
           that map to cache set $i$
       $samples$ : the number of timing measurements to be
       made (typically around $2^{20}$)
Output: $True$ : $\mathbb{E}$ is an eviction array else $False$: $\mathbb{E}$ is not an
           eviction array
1  begin
2      for $iteration \leftarrow 1$ to $(samples)$ do
3          Choose an address $x$ from $\mathbb{E}$.
4          Access $x$.                          // cold miss at $x$
5          Access $x$ again and measure access time $t_1$.
                                               // cache hit at $x$
6          Access all other members of $\mathbb{E}$.      // To evict $x$
7          Access and measure the access time $t_2$ of $x$.
8      if $Average(t_2 - t_1) \geq threshold$ then
9          return $True$            // $\mathbb{E}$ is an eviction array
10     else
11         return $False$   // $\mathbb{E}$ is not an eviction array
```

To determine a conflict array for a cache set, we define a large array that uses 2MB huge pages. The huge pages will ensure that the set index bits of the physical address can be observed from the corresponding virtual address [17]. To build a conflict array for the $i$-th cache set, we need to find $M$ eviction arrays. Algorithm 1 shows how an eviction array can be identified. The input is a set $\mathbb{E}$ of at least $w+1$ addresses picked from the huge array that map to the $i$-th cache set, where $w$ is the associativity of the LLC. If these $w+1$ addresses form an eviction array, they will fall in the same slice and conflict. Algorithm 1 identifies a conflict by an increase in memory access time and will return *True*. Otherwise it will return *False*.

Algorithm 1 is repeated multiple times until $M$ *non-conflicting* eviction arrays are found. This would form the conflict array $\mathbb{C}_i$ for cache set $i$. The non-conflicting requirement ensures that the $M$ eviction arrays correspond to different slices of the cache set. This conflict array is valid till the system is rebooted. Upon reboot, the huge array may get mapped to a different physical address, therefore invalidating the conflict array. It is therefore essential to optimize this algorithm. We refer the reader to [17] for a more optimized algorithm. For this work we have evaluated two CPUs: a four core Intel Core i7-3770 CPU and a 10 core machine Intel E5-2640 v4 CPU. Optimized version of the algorithm from [17] was used for the purpose.

**Scout Phase.** The objective of the scout phase is to determine the LLC cache sets that the victim occupies. We are especially interested in the cache sets that would leak sensitive information about the victim. The process of determining these cache sets, which we denote as *interesting cache sets*, depends on the attacker's knowledge about the victim process. If the attacker knows the victim's physical address

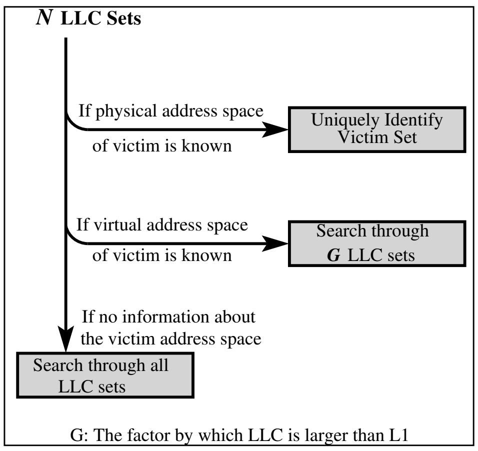

N LLC Sets
If physical address space of victim is known
Uniquely Identify Victim Set
If virtual address space of victim is known
Search through G LLC sets
If no information about the victim address space
Search through all LLC sets
G: The factor by which LLC is larger than L1

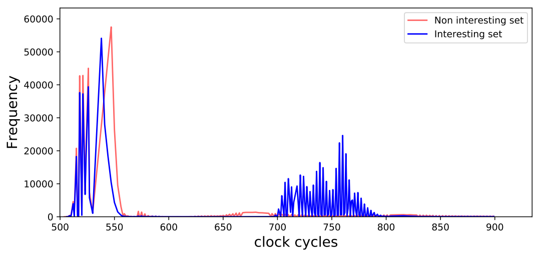
Fig. 2: Scout phase complexity depends on the attacker's knowledge about the victim process' address space.

map, she can uniquely determine the LLC cache sets that the victim will occupy from the set index bits in the victim's physical address. On the other hand, if the attacker only knows the victim's virtual address map, she would only be able to identify the L1 cache sets that the victim occupies. Each L1 cache set can be mapped from $G$ LLC cache sets; where $G$ is the factor by which the LLC is larger than the L1 cache. For example, with a $32KB$ L1 data cache and a $2MB$ LLC slice, $G$ is 32. This means that, with knowledge of the victim's virtual address space, the attacker would need to search through 32 LLC cache sets to identify the interesting ones. If neither the victim's virtual address map nor the physical address map is known, then the attacker would need to search through all LLC cache sets in order to identify the interesting sets. These alternatives are depicted in Figure 2. In this paper, we consider the latter category, where the attacker neither knows the victim's virtual address map nor physical address map and therefore would require to search the entire LLC to find the interesting cache sets.

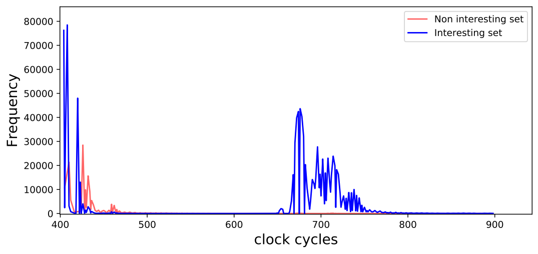
To determine if an LLC cache set $i$ is interesting, the attacker executes the victim continuously and also runs a spy in parallel, which continuously executes load operations to memory locations determined by the conflicting array $\mathbb{C}_i$. If the victim is indeed accessing the cache set $i$, then there would be an interference between the victim and the spy processes due to the shared cache set. This leads to an increase in execution time of the victim process as well as the spy process.

As an example, consider the AES implementation, which uses four T-tables, each of 1024 bytes[2]. Each T-table spreads over 16 LLC cache sets. These are the interesting cache sets, which the attacker must locate. Assuming that neither the virtual nor physical addresses of these T-tables are known to the attacker, she would have to search through the entire LLC in order to find the cache sets that the T-tables

[2] OpenSSL ver. 1.0.1f (https://www.openssl.org/)



(a) CPU: 4-core Intel $i7-3770$



(b) CPU: 10-core Intel E5-2640 v4

Fig. 3: Frequency distribution of execution time at the victim for interesting and non-interesting cache sets. With $2^{19}$ execution time measurements, the interesting cache sets can be identified with an accuracy of 96% when victim execution time is measured.

occupy. To achieve this, she would first find the conflicting arrays for all the LLC cache sets as described in the learning phase. In the scout phase, for each cache set, she would access the conflicting array continuously and monitor the execution time of the AES encryption or the memory access time in the spy process. An increased memory access time would indicate that the set is potentially interesting. Figure 3 shows the frequency distribution of the execution time for an interesting cache set (blue) and a non-interesting cache set (red) on a 4 core Intel(R) Core(TM) i7-3770 CPU with $2^{19}$ encryptions of victim. We found that measuring the time at the victim (Figure 3) led to significantly higher accuracy in identifying the interesting cache sets. Figure 4 shows the frequency of obtaining high execution time when the spy was accessing a particular LLC cache set of a 10 core Intel CPU. The graph is plotted after making $2^{19}$ victim AES encryptions for each cache set. The 64 interesting cache sets for AES are marked in blue in the figure.

**Strike Phase.** If the victim makes memory accesses that are dependent on its secret information, then the cache timing can be made to leak information about the secret. We explain the strike phase with AES as an example. The AES implementation we consider, accesses 1024 byte T-tables at locations that depend on the plaintext and secret key. In particular, the first round of AES accesses the T-tables at locations given by $p \oplus k$, where $p$ is a plaintext byte and $k$ is a byte of the secret key. Cache-timing attacks are feasible on this implementation because the locations of the T-tables accessed during encryption depends on the secret key.

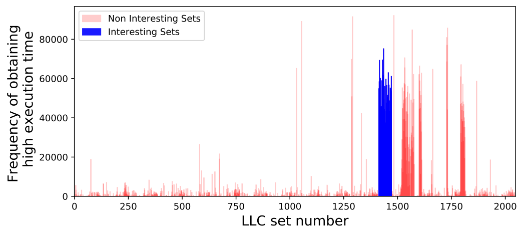


Fig. 4: Frequency of obtaining high execution time in victim vs cache set accessed by spy (CPU: 10-core Intel E5-2640 v4).

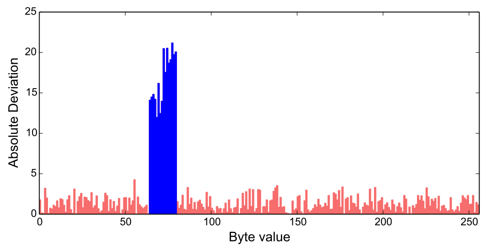


Fig. 5: Average execution time deviation vs value of a plaintext byte $p_i$. The maximum deviation regions (in blue) correspond to collisions with the spy. These regions would provide bits of the secret key. (CPU: 10-core Intel E5-2640 v4)

**Algorithm 2:** Strike phase to obtain secret key byte $k_i$

```
   Input: $q$- targeted cache set in a T-table
   Output: Guessed key byte $k_i$
1  begin
       Init: $total\_time$ : array of size 256, initially empty
       Init: $count$ : array of size 256, initially empty
2
       /* Invoke $N$ encryptions and build timing
          profile                                  */
3      for $iterations \leftarrow 1$ to $N$ do
4          $PT \leftarrow (p_0||p_1||\cdots||p_{m-1})$ where $p_j \xleftarrow{R} \{0,255\}$ for
             $0 \le j \le m-1$
5          $(CT,t) \leftarrow \mathscr{E}_k(PT)$
6          $total\_time[p_j] = (total\_time[p_j]+t)$ where
             $0 \le j \le m-1$
7          $count[p_j] = (count[p_j]+1)$ where $0 \le j \le m-1$
8
       /* Compute execution time deviations in
          array $D$                                 */
9      $avg[k] = \frac{total\_time[k]}{count[k]}$ for $0 \le k \le 255$
10     $Average = \frac{\sum_{r=0}^{255} total\_time[r]}{\sum_{r=0}^{255} count[r]}$
11     $D[k] = |Average - avg[k]|$ for $0 \le k \le 255$
12
       /* Obtain the $byteValue$ with maximum
          deviation                                */
13     $byteValue = \arg\max_i D_i$          // find the $i$
         corresponding to maximum deviation
14     return $(byteValue \oplus q)$
```

To obtain bits of the secret key, we develop an Evict+Time based cache attack. The attacker first uses the scout phase to find the LLC cache sets that hold the T-tables. There are 16 such interesting cache sets per table. She picks one (say the $q$-th $(0 \le q \le 15)$) interesting cache set, which we denote as $I_t$, and continuously accesses the corresponding conflict array in the spy process. If $p \oplus k$ happens to access the cache set $I_t$, it would result in a higher execution time of the AES victim due to conflicts with the spy. We detect this increased execution time and use $q$ to determine bits in the secret key byte $k_i$.

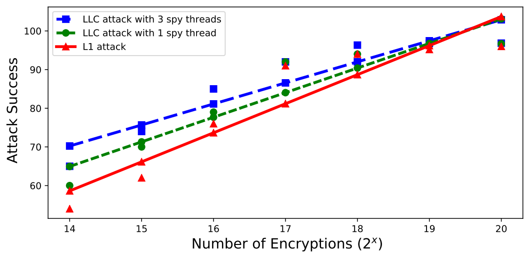
Algorithm 2 has the complete attack. The attacker invokes the victim encryption routine for a large number of times (Lines 3 to 7). In each iteration of the loop, the attacker chooses a random plaintext $PT$, of $m$ bytes and invokes the AES encryption $\mathscr{E}_k$ to obtain the ciphertext $CT$. Additionally, the time taken $t$, to perform the encryption is measured and logged in an array called $total\_time$ at the index $p_i$ (Line 6). Also the number of times $p_i$ occurred is recorded in the array $count$ (Line 7). After the loop, the deviation from the average execution time is computed. The highest deviated value, corresponds to conflicts with the targeted cache set $I_t$. The corresponding plaintext byte $byteValue$ and the index in the T-table $q$ that falls in the same cache set $I_t$ are ex-ored to compute the value of key byte $k_i$.

Figure 5 plots the absolute deviations of execution time for the victim with various values of the plaintext byte $p_i$. The blue lines in the graph corresponds to collisions with the targeted LLC cache set $I_t$. The value on the x-axis corresponding to the maximum deviation is obtained as $byteValue$ and used to compute the secret key byte $k_i$. LLC attacks offer several avenues to improve the success. In the next section, we describe optimization techniques for the strike phase, while Section 6 describes optimization techniques for the scout phase.

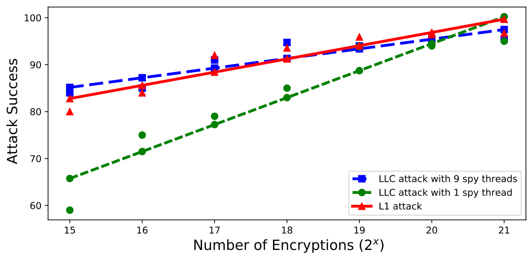
## 5 Optimizing the Strike Phase

The success of a cache-timing attack depends on the amount of interference between the spy and the victim processes. If the spy can evict the interesting cache lines at a higher rate, there would be an increase in interference with the victim process, leading to a higher attack success. Typically, L1 cache attacks are expected to be more powerful because of the quick memory accesses that are possible. However, LLC attacks have two distinct advantages over L1 cache attacks. First, the spy process in an LLC attack runs on an independent CPU core, which can be made to be less loaded compared to a spy in the L1 cache attack, that shares the CPU core with the victim. Due to this, the spy in an LLC attack executes much faster than the spy in an L1 cache attack. This leads to higher interference between the spy and victim processes and hence a better attack success.

Second, an attacker may be able to run a spy in every core of the CPU. For instance, suppose a victim is executing in one of the cores of a 4 core CPU, the attacker can run a spy thread in each of the remaining 3 CPU cores. Each of these



(a) CPU: 4-core Intel $i7-3770$



(b) CPU: 10-core Intel E5-2640 v4

Fig. 6: Average attack success vs number of time measurements (encryptions) in the strike phase with different spy configurations.

spy threads, would target the same interesting cache line. This increases the eviction rate for that cache line, further increasing the interference and therefore the attack success.

Figure 6 compares the attack success for the Evict+Time based cache attack on AES on two CPUs with the number of time measurements made ($N$) in Algorithm 2. Each graph compares the L1 cache attack with a single threaded LLC cache attack and a parallel version. In the 4-core CPU (Figure 6a), both the LLC attacks perform better than the L1 cache attack. In the 10-core CPU (Figure 6b), the attack with a spy having 9 threads, performs as good as the L1 cache attack.

## 6 Optimizing the Scout Phase

The purpose of the scout phase is to find LLC cache sets that are occupied by the victim process. Assuming that no information about the physical address space of the victim is known (ref. Figure 2), the attacker would require to search through all the cache sets in the LLC to identify the interesting victim cache sets. This search process would take a long time as the LLC is considerably large. Further, the scout phase could have false negatives, where interesting cache sets are not identified. In this section, we describe optimization techniques to reduce the false negatives and the runtime of the scout phase.

### 6.1 Searching for Interesting Sets

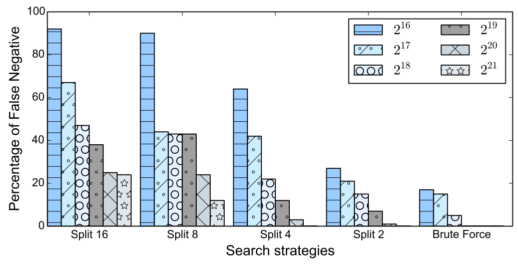
One option to optimize the scout phase is to do a binary search for the interesting sets. For an L1 cache set (say the $i$-th set), a binary search is done on the LLC cache sets that map to this $i$-th L1 set. We denote these LLC cache sets as $\mathbb{G}_{L1_i}$. For example, assuming that the L1 cache has 64 sets, and there are 2048 sets in a slice, $\mathbb{G}_{L1_0}$ will have the LLC cache sets $0, 64, 128, 192, \cdots, 1984$. We assume that each $\mathbb{G}_{L1_i}$ has at most one interesting cache set. This is generally the case as the victim's sensitive data is generally clustered together in contiguous cache sets. For example, in the AES T-table implementation, each T-table maps to 16 contiguous cache sets.

For determining if any interesting cache sets are present in $\mathbb{G}_{L1_i}$, we can perform a binary search. In each step of the binary search, the spy process would continuously access all conflicting arrays of a subset of $\mathbb{G}_{L1_i}$. To start off, the spy would sequentially access half of the conflicting sets in $\mathbb{G}_{L1_i}$. While the spy executes, the victim process is also executed in parallel in a different CPU core and its average execution time is measured. If this execution time exceeds a threshold, we denote that there is an interesting set and perform the next step of binary search on a smaller subset of $\mathbb{G}_{L1_i}$. This continues until the interesting set, if present, is uniquely identified.

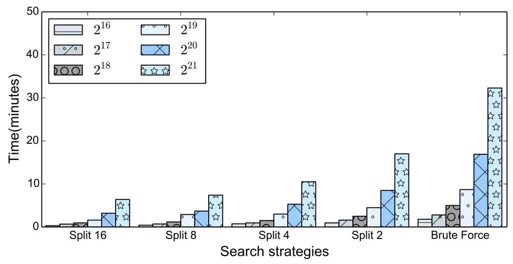
The advantage of a binary search is that it would take the shortest time to search through the entire LLC. However, there would be considerable number of false negatives. The false negatives are especially prominent at the start of the binary search, where the spy sequentially accesses a larger number of conflicting arrays. This reduces the eviction rate per LLC cache set thereby reducing the accuracy of identifying the interesting sets.

An alternate approach is a linear search technique, where the spy continuously accesses one cache set in $\mathbb{G}_{L1_i}$ at a time. This reduces false negatives but would increase runtime for the scout phase. The best approach we found is the one in which the spy process accesses a fixed small subset of $\mathbb{G}_{L1_i}$ at a time. This approach provides a good trade off between false negatives and the runtime for the scout phase. For example, the spy continuously accesses four cache sets of $\mathbb{G}_{L1_i}$ at a time. We call this the *Split 4* search. If an increase in execution time in the victim process is observed, we infer that one of these four cache sets is interesting. We then perform a binary search on these four cache sets to uniquely identify the interesting set.

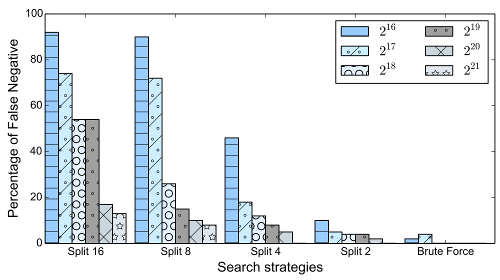
The accuracy of the result and runtime for the scout phase varies with the value of $N$. An increase in $N$ improves accuracy of identifying interesting victim sets, but takes a longer time to execute. Figure 7 compares the scout phase runtime and false negatives with different search strategies (*Split* $s_z$) and values of $N$, where $s_z$ is a power of two, typically 4 or 8. The percentage of false negatives varies directly with the number of LLC cache sets probed simultaneously ($s_z$) and it varies inversely with $N$. On the other hand, the runtime varies inversely with $s_z$ and varies directly with $N$.



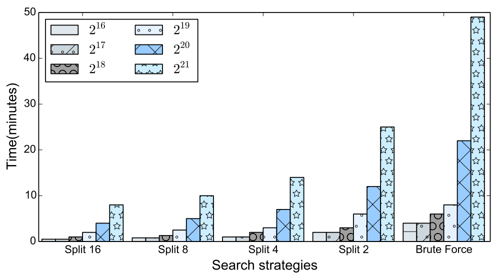
(a) Percentage False Negatives vs Search Strategy on Intel 4-core i7-3770.



(b) Runtime vs Search Strategy on Intel 4-core i7-3770.



(c) Percentage False Negatives vs Search Strategy on 10-core Intel E5-2640 v4.



(d) Runtime vs Search Strategy on 10-core Intel E5-2640 v4.

Fig. 7: The graphs show search strategy vs percentage of false negatives and search strategy vs the scout phase runtime for different values of $N$. As $N$ increases, runtime increases but false negatives reduce. The number of cache sets probed (search strategy) at a time directly affects the false negatives and inversely affects the runtime. A good trade off between the two, is for search strategies *Split 4* and *Split 8*.

### 6.2 Scout Phase with Multiple Threads

As described in Section 5, LLC attacks have the advantage of parallelizing the spy and executing one spy in each core. In an $M$ core CPU, the attacker can create $M-1$ spy threads; assuming at-most one spy (or victim) runs in each core as seen in Figure 1. The $M-1$ spy threads can be configured to either reduce the time for the scout phase or increase the accuracy in detecting interesting cache sets.

**Increasing Accuracy.** Accuracy in detecting interesting cache sets is affected by eviction rate of the cache sets shared between victim and spy. A lower eviction rate would imply that the victim is less affected by the spy. As a result, its execution time is not affected much. Multiple spy threads targeting the same cache set would increase the eviction rate, which would make detection of interesting cache sets more accurate.

**Reducing Time.** In order to reduce runtime for the entire scout phase, the LLC cache sets can be partitioned (P0, P1, ..., P9, in Figure 1). Each partition can then be searched in parallel by different spy threads. For example, in a 10 core CPU, the attacker can divide the LLC sets into 9 partitions. Apart from the CPU core holding the victim, spy threads on all the remaining cores will get assigned a different partition of the LLC. For an LLC with 2048 cache sets, each spy thread would only need to search about 228 (2048/9) cache sets. This will reduce the runtime for the scout phase by approximately one ninth. However, with this scheme, the accuracy with which interesting cache sets are found, will be reduced. In fact, the accuracy will be worse compared to a spy that is not parallelized. This is because, the 9 spy threads would be making memory accesses at a high rate to different memory locations. We would also have the victim executing. There would be contention for the shared resources, namely the front-side ring bus and the shared LLC. Of the 9 spy threads, only one thread could be accessing an interesting cache set at a given time. Due to contention of the shared resources, the eviction rate of the victim would reduce leading to a reduced accuracy in detecting the interesting cache sets. We discuss this further and provide results in Section 8.

**Trade off Between Accuracy and Time.** Instead of partitioning the cache into nine parts, the cache can be partitioned in many other ways. For example, the cache can be partitioned into 3, with 3 spy threads assigned to each partition. Compared to having 9 partitions, this scheme would increase accuracy (lower false negatives) in determining interesting cache sets, but would also have higher runtime. For a 10-core CPU, with a spy having 9 threads, there are 253 different configurations. Each configuration would provide a different accuracy and would have different runtime. Some configurations would have high accuracy but take longer, while other configurations would have lower accuracy but complete faster. In the next section we discuss an attack methodology, by which an attacker can choose a spy configuration, depending on the available time-frame, so as to maximize the attack success.

## 7 Spy Configuration for Maximizing Attack Success

All cryptanalytic attacks are time bounded. An attack is only considered successful if it can retrieve the secret key within practical time bounds. For side-channel attacks in particular, these bounds are more critical because the attacker needs access to the device performing the cryptographic operations. For cache-timing attacks, the time bounds are also important to elude detection. In this section we present a new attack methodology by which an attacker can maximize her success for a given time bound. To maximize success for a given time bound, the attacker would have to choose a spy configuration, which provides best success. If she optimizes for time, by maximally partitioning the LLC, the attack's success may by reduced. If she optimizes for success, by configuring all spy threads for the same cache set or increasing the encryptions ($N$ for the scout phase and in Algorithm 2 for the strike phase), she may not complete the attack within the given time-frame. She therefore has to choose a spy configuration that would allow her to maximize success and also ensure that the attack is completed. In this section we propose a methodology by which an attacker can decide apriori, the optimal spy configuration. The only requirement is a marginal increase in the offline learning. No modifications are required in the online scout or strike phases.

### 7.1 Spy Configurations

In an $M$ core CPU, we assume at-most $M-1$ spy threads. These $M-1$ spy threads can be assigned to $P$ partitions in the LLC, where $1 \leq P < M$. A *spy configuration* is a function that maps the $M$ threads to the $P$ partitions. Each spy configuration is denoted as follows:

$$C(a_1, a_2, \cdots, a_P) \ ,$$

where $a_1$ , $a_2$, $\cdots$ , $a_P$ denote the number of spy threads assigned to each partition. Note that $a_1 + a_2 + \cdots + a_P \leq M-1$ and $a_i \geq 1$, $(1 \leq i \leq P)$. For example, a spy configuration for $M = 10$ and $P = 2$, is $C(5,4)$. This configuration means that the LLC of the 10-core CPU is partitioned into two, the first partition has 5 spy threads, while the second partition has 4 spy threads assigned to it. Similarly there are 7 more spy configurations with $P = 2$: $C(8,1)$, $C(1,8)$, $C(7,2)$, $C(2,7)$, $C(6,3)$, $C(3,6)$, and $C(4,5)$. Our proposed attack methodology cannot distinguish between *permuted spy configurations* such as $C(8,1)$ and $C(1,8)$. In this example, we therefore only consider 4 spy configurations instead of 8.

For each spy configuration that we consider, there are two parameters that impact the success of the attack and the runtime of the scout phase. First is the number of partitions $P$, while the second is the number of victim encryptions $N$ made in the strike phase (Algorithm 2) and scout phase. An increase in $P$ would imply shorter runtime but lower accuracy of detecting interesting cache sets. An increase in $N$ would imply longer runtime but higher accuracy. In Section 7.2, we introduce a metric to evaluate a spy configuration.

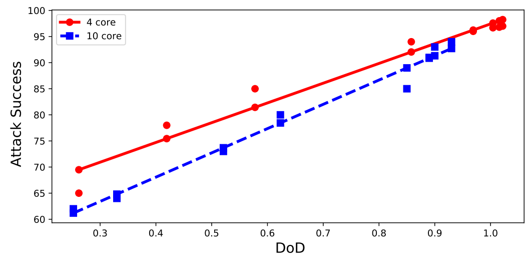


Fig. 8: The graph shows that there is a linear relationship between DoD and the attack success (CPUs: 10-core Intel E5-2640 v4 and 4-core Intel i7-3770).

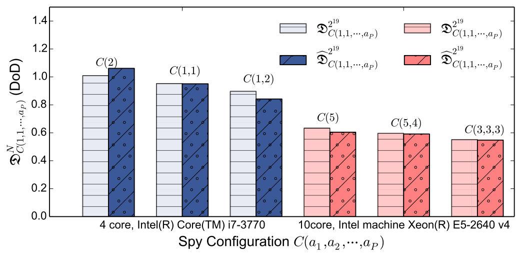


Fig. 9: This figure compares the theoretically estimated DoD with the empirical DoD for few spy configurations in two CPUs. For each case $N = 2^{19}$.

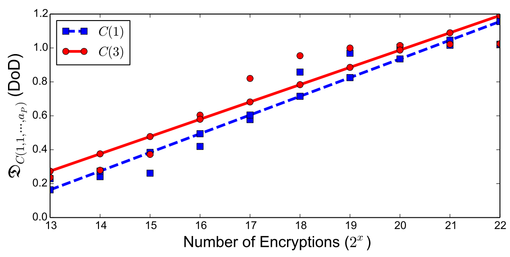
### 7.2 Evaluating Spy Configurations

Algorithm 2, Line 11, computes the average deviation in execution time of the victim under different inputs. The input corresponding to the highest deviation is used to compute the victim's secrets (the secret key byte in Algorithm 2). These deviations are also observed in Figure 5, where the blue indicates the highest deviations, corresponding to conflicts with the interesting cache sets, and the red indicates all other deviations. These deviations can be used as an indicator of the attack success. Higher the blue deviation compared to the red, higher the attack success. To capture these deviations, we define a metric, based on test vector leakage assessment(TVLA) [10] as follows:

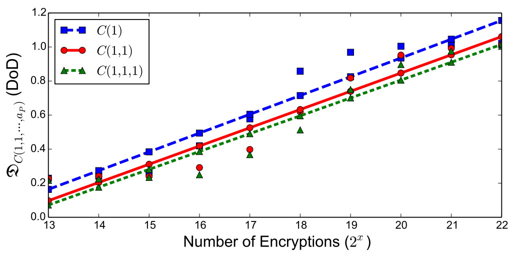
$$\mathfrak{D} = \frac{|X_b - X_r|}{\sqrt{\frac{{S_b}^2}{N_b} + \frac{{S_r}^2}{N_r}}} \quad . \tag{2}$$

Here $X_b$ and $X_r$ are respectively the means of deviations corresponding to the blue and red regions in Figure 5 while $S_b$ and $S_r$ represent the corresponding standard deviations, and $N_b$ and $N_r$ are the corresponding number of points. We denote this metric as the *Deviation of Deviations* (DoD).

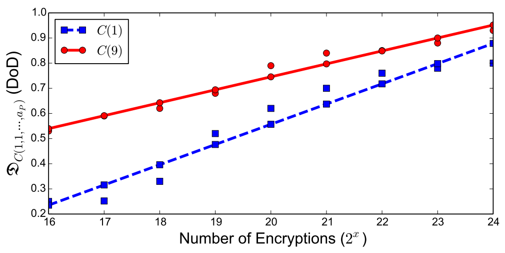
Figure 8 shows that there is a linear relationship between attack success and $\mathfrak{D}$. Thus, this metric is a good indicator of the attack success and can be used to evaluate spy configurations. Figure 10 plots this metric for different spy configu-



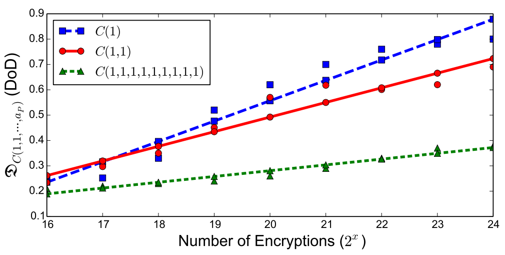
(a) Single Partition, Multiple spies on 4-core Intel i7-3770



(b) Multiple partition on CPU 4-core Intel i7-3770



(c) Single Partition, Multiple spies on CPU: 10-core Intel E5-2640 v4



(d) Multiple partition on CPU 10-core Intel E5-2640 v4

Fig. 10: DoD vs number of encryptions for different spy configurations. DoD is higher if the number of spy threads targeting the same partition is higher or $N$ is increased. DoD reduces as the number of partitions increases.

rations in a 4-core and a 10-core CPU. We list the inferences from these figures:

- As the number of spy threads targeting the same cache set increases, $\mathfrak{D}$ will also increase (Figures 10a and 10c). Therefore the attack success will also increase.
- If the number of partitions increases, then $\mathfrak{D}$ decreases due to flooding of the front side ring bus and the LLC (Figures 10b and 10d).
- As $N$ increases, $\mathfrak{D}$ increases and eventually saturates. After a large number of encryptions (*i.e.* large $N$), the different spy configurations will result in same $\mathfrak{D}$, therefore similar attack success. This is evident from Figures 10a and 10c, where the trends due to different spy configurations would eventually intersect.
- As one would expect, since the 4-core CPU can only have 3 spy threads, the benefits obtained is lesser compared to the 10-core CPU, with 9 spy threads (Figures 10a and 10c). We also observe that the 10-core CPU is more affected with higher number of partitions compared to the 4-core CPU (Figures 10b and 10d).

Thus $\mathfrak{D}$ is a function of the number of victim encryptions ($N$) and the spy configuration. For a given $N$ and spy configuration, $C(a_1, a_2, \cdots, a_P)$, we denote the corresponding metric as $\mathfrak{D}^N_{C(a_1,a_2,\cdots,a_P)}$. In Section 7.3, we propose an algorithm to estimate this metric.

### 7.3 Apriori Estimation of $\mathfrak{D}$ for a Spy Configuration

Given a spy configuration $C(a_1, a_2, \cdots, a_P)$ and the number of encryptions $N$, we devise an algorithm to estimate $\mathfrak{D}^N_{C(a_1,a_2,\cdots,a_P)}$. We denote this estimate as $\widehat{\mathfrak{D}}^N_{C(a_1,a_2,\cdots,a_P)}$. The algorithm for an $M$-core CPU is as follows.

1. For the given $N$, we first determine the DoD for the following 3 spy configurations using Algorithm 2: $C(1)$, $C(M-1)$, $C(1,1,1,\cdots(M-1)\text{ times}))$. These DoDs, which are determined empirically, are denoted as $\mathfrak{D}^N_{C(1)}$, $\mathfrak{D}^N_{C(M-1)}$, and $\mathfrak{D}^N_{C(1,1,1,\cdots(M-1)\text{ times})}$ respectively. All three configurations use Equation 2 to compute the DoD. The first two spy configurations, compute the DoD with minimum spy threads (*i.e.* one) and maximum spy threads (*i.e.* $M-1$) respectively. In both cases, the number of partitions is 1, and the spy threads simultaneously target the same cache set. This captures (approximately) the entire range of possible DoD values. The third spy configuration, maximally partitions the LLC. This captures the impact of flooding of the shared front side bus and LLC on the DoD.
2. Given these three DoDs, we can estimate DoDs $\widehat{\mathfrak{D}}^N_{C(2)}$, $\widehat{\mathfrak{D}}^N_{C(3)}$, $\cdots$, $\widehat{\mathfrak{D}}^N_{C(M-2)}$ using the following formulation:

$$\widehat{\mathfrak{D}}^N_{C(i)} = \mathfrak{D}^N_{C(1)} + (i-1) \times \frac{\mathfrak{D}^N_{C(M-1)} - \mathfrak{D}^N_{C(1)}}{M-2} \quad , \qquad (3)$$

where $2 \le i \le M-2$. These estimated DoDs correspond to a number of spy threads targeting a single partition. This estimation is based on the assumption that the DoD varies linearly with the number of spy threads. The component

$$(i-1) \times \frac{\mathfrak{D}^N_{C(M-1)} - \mathfrak{D}^N_{C(1)}}{M-2},$$

represents the increase in the DoD with $i-1$ additional spy threads, compared to a single-threaded spy $C(1)$.

3. Next, we estimate the DoD with increasing partitions using the following formulation:

$$\widehat{\mathfrak{D}}^N_{C(1,1,\ldots(i \text{ times}))} = \mathfrak{D}^N_{C(1)} - (i-1) \times \frac{\mathfrak{D}^N_{C(1)} - \mathfrak{D}^N_{C(1,1,\ldots,(M-1 \text{ times}))}}{M-2} \quad (4)$$

where $2 \leq i \leq M-2$. These DoDs correspond to $i$ partitions of the LLC, with each partition having exactly one spy thread. The estimation is based on the assumption that the DoD varies inversely with the number of partitions in the LLC. The component

$$(i-1) \times \frac{\mathfrak{D}^N_{C(1)} - \mathfrak{D}^N_{C(1,1,\ldots,(M-1 \text{ times}))}}{M-2} \quad ,$$

represents the decrease in the DoD with $(i-1)$ additional partitions, and each partition assigned to one thread. The decrease in DoD is due to the contention for the shared front side bus and LLC.

4. Now, we can estimate the DoD for any spy configuration. We assume that the spy configuration partitions the LLC into $P$ partitions. Since all partitions are of equal size, the interesting set can be present in any of the partitions with equal probability. Let us first assume that the interesting set is in the $i$-th partition, which has $a_i$ spy threads attached to it. The estimated DoD obtained in this case is influenced positively by these $a_i$ threads, and negatively by the remaining spy threads, which target the other partitions. We further assume that the negative effect is only influenced by the number of spy threads targeting the $P-1$ partitions that do not contain the interesting set. Thus the estimated DoD for this $i$-th partition is $\left(\mathfrak{D}^N_{C(1)} + \widehat{\mathfrak{D}}^N_{C(a_i)} - \widehat{\mathfrak{D}}^N_{C(1,1,\ldots,(j \text{ times}))}\right)$, where $j = \sum_{k=1;k\neq i}^{P} a_k$ .
To find the estimated DoD considering all partitions, we compute each DoD independently and take the average as follows:

$$\mathfrak{D}^N_{C(a_1,a_2,\cdots,a_P)} = \frac{1}{P}\sum_{i=1}^{P}\left(\mathfrak{D}^N_{C(1)} + \widehat{\mathfrak{D}}^N_{C(a_i)} - \widehat{\mathfrak{D}}^N_{C(1,1,\ldots,(j \text{ times}))}\right)$$
$$\text{where } j = \textstyle\sum_{k=1;k\neq i}^{P} a_k \quad (5)$$

Figure 9 compares the estimated DoD that we compute with empirically obtained DoD for a few configurations in the 4-core and 10-core CPUs. Similar results are obtained for all other spy configurations as well.

## 7.4 **Apriori Estimation of Runtime for a Spy Configuration**

Runtime for the scout phase depends on the number of cache sets that need to be searched and the time taken to determine if a cache set is interesting. The latter depends on the number of timing measurements made, which we denote by $N$. An increase in the number of timing measurements, will increase the runtime, but also increases the accuracy of detecting interesting cache sets. If the cache sets are partitioned into $P$ partitions, where $1 \leq P \leq M-1$, then time for the scout phase reduces by a fraction of $P$.

In order to estimate the runtime for the scout phase for a given spy configuration, the following steps are followed:

1. For different $N$ values, we execute the spy configuration $C(1)$ and record the time taken for the entire scout phase. We choose this configuration because it is the fastest single spy thread configuration.
2. Again, for different $N$ values, we execute the spy configuration $C(M-1)$, where $M$ is the number of cores in the CPU and record the time taken for the entire scout phase. We choose this configuration because it is the slowest single-threaded spy configuration.
3. We find the average scout phase time considering $C(1)$ and $C(M-1)$, for all the different values of $N$.
4. Now, for a given $N$ and any spy configuration with $P$ partitions, we can approximate the scout time as $1/P$ of the time computed in step 3.

## 7.5 **Methodology for Configuring Parallel Spy Threads**

We now present the attack methodology by which an attacker can choose a spy configuration and a value of $N$ that would maximize success given a time-frame $T_f$. This choice is made offline during the learning phase. The chosen spy configuration is then used during the scout and strike phases of the attack. The attack methodology comprises of three steps as follows:

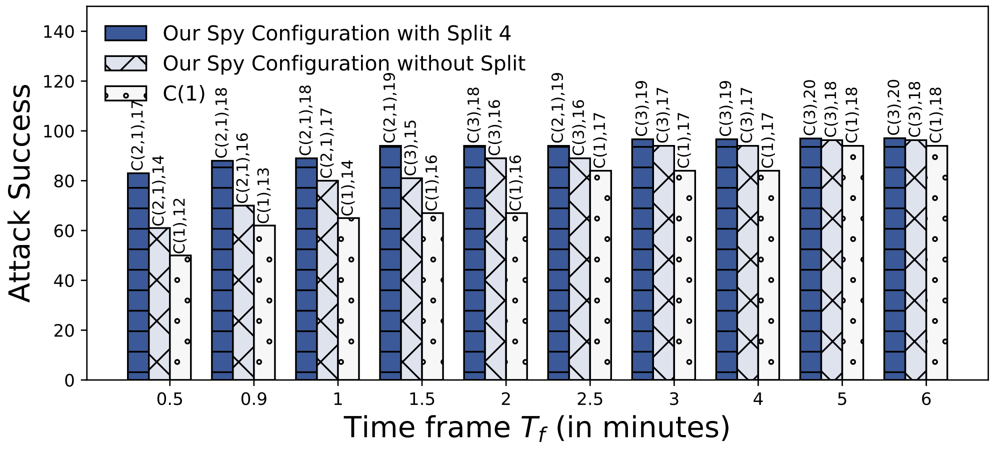
1. For every possible spy configuration, we use the steps in Section 7.4 to determine the maximum value of $N$ for which the entire scout and strike phase can be completed within the giving time-frame $T_f$.
2. For these spy configurations and their respective values of $N$, we estimate the DoD as discussed in Section 7.3.
3. We select the spy configuration with the highest DoD. This spy configuration is used during the scout and strike phases of the attack.

## 8 Results

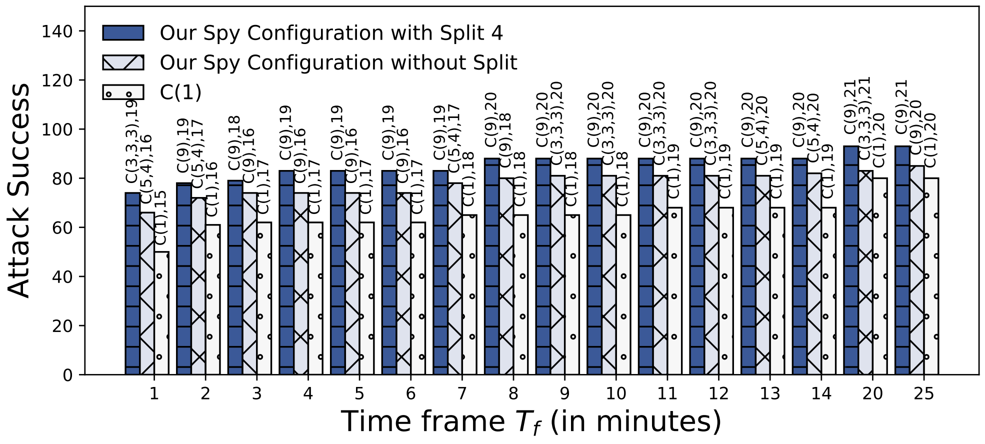
In this section, we compare our attack methodology, with a naïve Evict+Time based attack technique. The victim is an OpenSSL (ver. 1.0.1f) implementation of AES. We assume that the attacks must first probe the entire LLC during the scout phase and then determine the AES secret key in the strike phase. The naïve implementation is the spy configuration $C(1)$ has a single-threaded spy. For a given time-frame, we set the value of $N$ (ref. Algorithm 2 and Figure 11) for this configuration such that the entire LLC is probed (*i.e.* scout phase must be completed) and the attack's success is maximum. For instance in a 4-core CPU, if $T_f$ is 1.5 seconds, $N$ must be $2^{16}$. If $N < 2^{16}$, the attack's success is reduced, while if $N > 2^{16}$, the scout phase does not complete.

Table 1: Overheads (in minutes) for the learning phase due to the proposed attack methodology in two CPUs: M=4 (4-core Intel i7-3770) and M=10 (10-core Intel E5-2640 v4)

| $M$ | Spy Configuration | Number of Encryptions ($N$) | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $2^{13}$ | $2^{14}$ | $2^{15}$ | $2^{16}$ | $2^{17}$ | $2^{18}$ | $2^{19}$ | $2^{20}$ | $2^{21}$ | $2^{22}$ | $2^{23}$ | $2^{24}$ |
| **4** | $C(1)$ | 0.8 | 0.9 | 1.1 | 1.2 | 2.5 | 4.5 | 8.1 | 15.9 | 29 | - | - | - |
| | $C(3)$ | 0.9 | 1 | 1.2 | 1.8 | 2.7 | 4.7 | 8.9 | 16.7 | 32 | - | - | - |
| | $C(1,1,1)$ | 0.3 | 0.3 | 0.4 | 0.6 | 0.9 | 1.6 | 2.9 | 5.6 | 10.6 | - | - | - |
| **10** | $C(1)$ | - | - | - | 1.59 | 2.12 | 5.3 | 9.54 | 19 | 36 | 72 | 145 | 290 |
| | $C(9)$ | - | - | - | 2.12 | 3.2 | 6.4 | 12.7 | 24.4 | 49.8 | 97.5 | 195 | 392 |
| | $C(1_{(9\text{times})})$ | - | - | - | 0.24 | 0.36 | 0.71 | 1.4 | 2.7 | 5.5 | 10.8 | 21.6 | 43.6 |



(a) CPU: 4-core Intel i7-3770

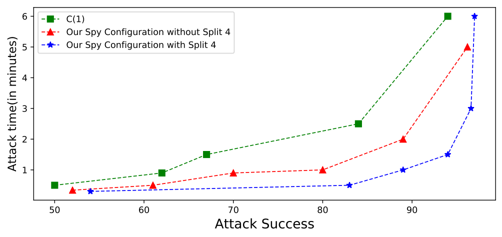


(b) CPU: 10-core Intel E5-2640 v4

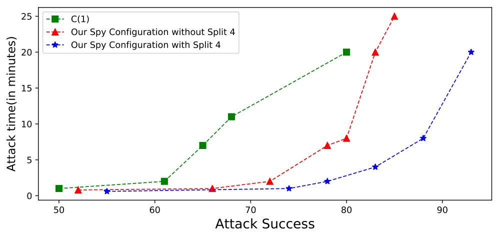
Fig. 11: Success of the attack vs timeframe of our attack methodology and the single-threaded attack ($C(1)$). Each spy configuration and $\log_2 N$ is also shown for each attack. The results show that our *Split 4* configurations always provides the highest attack success. The profits are more visible for small timeframes.

If the time-frame increases, then $N$ can also increase. This would improve the attack's success.

Our attack implementation has an additional step during the offline learning phase. In this step we compute $\mathfrak{D}^N_{C(1)}$, $\mathfrak{D}^N_{C(M-1)}$, and $\mathfrak{D}^N_{C(1,1,1,\cdots(M-1)\text{ times}))}$ for different $N$. These DoD values are then used to select optimal attack strategies for various time-frames as discussed in Section 7.5. Each choice comprises of a spy configuration and a value of $N$. $C(1,\cdots,_{(M-1)\text{times}})$ is the least because each thread only probes about $1/(M-1)$ sets of the entire LLC. The spy configuration $C(M-1)$ is slowed down due to contention for the shared front side bus and the LLC cache sets. Figures 11 compares the online phase of our attack with the single threaded spy configuration ($C(1)$) for different time-frames ($T_f$). For example, in the 4-core CPU, $T_f = 1.5$ indicates



(a) CPU: Intel 4-core i7-3770.



(b) CPU: 10-core Intel E5-2640 v4

Fig. 12: The graphs compares the attack success vs attack time. Higher success at lower attack time is better.

that the time taken for the entire online phase (scout+strike) is under 1.5 minutes. Without any split (ref. Section 6.2), our attack determines the secret key with an average success of 81%. The best attack we find, comprises of the spy configuration $C(3)$ and having a value of $N = 2^{15}$. This means that the LLC has only one partition and three spy threads probing the same cache set simultaneously.

When a *Split 4* search strategy is used, the attack determines the secret key with a success of about 94% for $T_f = 1.5$. The attack uses a spy configuration $C(2,1)$. This means that the LLC is partitioned into two. Two spy threads probe one partition, while the third spy thread probes the other partition. Since each spy thread needs to probe only half the sets in the LLC, $N$ can be increased from $2^{15}$ to $2^{19}$. All our attacks have a much higher success compared to the single-threaded spy, which has a success of about 67%. Notice that the *Split 4* search always provides the highest attack success. The high success with the *Split 4* search is because the scout phase completes about 4 times faster than a search without any splits. This permits larger $N$ values to be chosen, resulting in better success.

As seen in Figure 11, the success of our attacks compared to the single-threaded attack is much more pronounced when the time-frame is small. This is due to the fact that $N$ has to be small in order that the single-threaded attack strategy completes the entire scout phase. A small $N$ results in poor attack success. As the time-frame increases, larger $N$ can be used, which increases the success further. The impact of our attack methodology is more useful for CPUs with larger number of cores. This can be seen in Figure 11, where, even after 25 minutes, the single-threaded attack does not provide as much success as our attack configurations.

Figure 12 shows a different perspective of the attack success and time. It highlights the minimum average time required to attain a certain level of success. As can be seen from the graphs, our attack methodologies achieve higher success at much lesser time.

## 9 Conclusions

In this paper, we demonstrate how a multi-threaded spy can be used to improve the attack success as well as reduce runtime. We propose an attack methodology, by which, the attacker can choose a spy configuration that would maximize success for a given time-frame. The methodology is especially useful when the time-frame is small. For instance, given a time-frame of 1 minute, a single threaded LLC attack may not complete, while our parallel attack with optimally configured spy threads, would complete and achieve a success of over 50%. The attack methodology only requires an additional offline step to characterize the effect of multiple spies executing simultaneously.

Our research found that the proposed methodology with optimally chosen parallel spy threads, is more beneficial in CPUs having large number of processing cores. Such CPUs are becoming the norm, especially for servers and cloud computing platforms. These machines are also most vulnerable to cache-timing attacks due to multi-user environments. The parallel spy LLC attacks further increases the threat to these systems.

## References

1. Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES (short paper). In *International Conference on Information and Communications Security*, pages 112–121. Springer, 2006.
2. Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, pages 225–242, 2007.
3. Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2010.
4. Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-vm attack on AES. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2014.
5. Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 591–604, 2015.
6. Daniel J Bernstein. Cache-timing attacks on AES. 2005. URL `http://cr.yp.to/papers.html#cachetiming`.
7. Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *Information Tech-*

*nology: Coding and Computing, 2005. ITCC 2005. International Conference on*, volume 1, pages 586–591. IEEE, 2005.

8. Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2006.
9. Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, pages 667–684, 2009.
10. Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, 2011.
11. David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 490–505. IEEE Computer Society, 2011.
12. Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.
13. Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.
14. Francois Koeune, Jean-Jacques Quisquater, and Jean-Jacques Quisquater. A timing attack against Rijndael. 1999.
15. Cédric Lauradoux. Collision attacks on processors with cache and countermeasures. In *WEWoRC 2005 - Western European Workshop on Research in Cryptology, July 5-7, 2005, Leuven, Belgium*, volume 74 of *LNI*, pages 76–85. GI, 2005. ISBN 3-88579-403-9.
16. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018.
17. Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 605–622, 2015.
18. James L Massey. Guessing and entropy. In *Information Theory, 1994. Proceedings., 1994 IEEE International Symposium on*, page 204. IEEE, 1994.
19. Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*, volume 9404 of *Lecture Notes in Computer Science*, pages 48–65. Springer, 2015.
20. Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *Selected Areas in Cryptography, 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers*, volume 4356 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.
21. Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at Bernstein's AES side-channel analysis. In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2006, Taipei, Taiwan, March 21-24, 2006*, page 369. ACM, 2006. ISBN 1-59593-272-0.
22. Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in Javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1406–1418. ACM, 2015.
23. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.
24. Colin Percival. Cache missing for fun and profit. *BSDCan 2005*, pages 1–13.
25. Chester Rebeiro and Debdeep Mukhopadhyay. Micro-architectural analysis of time-driven cache attacks: Quest for the ideal implementation. *IEEE Trans. Computers*, 64(3):778–790, 2015.
26. Chester Rebeiro, Debdeep Mukhopadhyay, Junko Takahashi, and Toshinori Fukunaga. Cache timing attacks on clefia. In *Progress in Cryptology - INDOCRYPT 2009, 10th International Conference on Cryptology in India, New Delhi, India, December 13-16, 2009. Proceedings*, volume 5922 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2009.
27. Chester Rebeiro, Mainack Mondal, and Debdeep Mukhopadhyay. Pinpointing cache timing attacks on AES. In *23rd International Conference on VLSI Design, 9th International Conference on Embedded Systems, Bangalore, India, 3-7 January 2010*, pages 306–311. IEEE Computer Society, 2010.
28. Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.
29. Kris Tiri, Onur Aciiçmez, Michael Neve, and Flemming Andersen. An analytical model for time-driven cache attacks. In *FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, volume 4593 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2007.
30. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.
31. Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2003.
32. Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M Swift. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security Symposium*, pages 913–928, 2015.
33. Zhang Xu, Haining Wang, and Zhenyu Wu. A measurement study on co-residence threat inside the cloud. In *USENIX Security Symposium*, pages 929–944, 2015.
34. Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 719–732. USENIX Association, 2014.
35. Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015.
36. Xin-jie Zhao and Tao Wang. Improved cache trace attack on AES and CLEFIA by considering cache miss and s-box misalignment. *IACR Cryptology ePrint Archive*, 2010:56, 2010.