Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls

Sarani Bhattacharya[1], Clementine Maurice[2], Shivam Bhasin[3], and Debdeep Mukhopadhyay[1,3]

Indian Institute of Technology Kharagpur
Graz University of Technology Austria
Nanyang Technological University Singapore

Abstract. In recent years, performance counters have been used as a side channel source for the branch mispredictions which has been used to attack ciphers with user privileges. However, existing research considers blinding techniques, like scalar blinding, scalar splitting as a mechanism of thwarting such attacks. In this endeavour, we reverse engineer the undisclosed model of Intel's Broadwell and Sandybridge branch predictor and further utilize the largely unexplored perf ioctl calls in sampling mode to granularly monitor the branch prediction events asynchronously when a victim cipher is executing. With these artifacts in place, we target scalar blinding and splitting countermeasures to develop a key retrieval process using what is called as *Deduce & Remove*. The *Deduce* step uses template based on the number of branch misses as expected from the 3-bit model of the BPU to infer the matched candidate values. In the *Remove* step, we correct any erroneous conclusions that are made, by using the properties of the blinding technique under attack. It may be emphasized that as in iterated attacks the cost of a mistaken deduction could be significant, the blinding techniques actually aids in removing wrong guesses and in a way auto-corrects the key retrieval process. Finally, detailed experimental results have been provided to illustrate all the above steps for point blinding, scalar blinding, and scalar splitting to show that the secret scalar can be correctly recovered with high confidence. The paper concludes with recommendation on some suitable countermeasure at the algorithm level to thwart such attacks.

Keywords: Scalar Multiplication, Scalar Splitting, Scalar Blinding, 3-bit predictor

1 Introduction

Micro-architectural side-channel threats have gained importance manifold in the last decade since the cloud service providers allow several users to share the same hardware. These attacks target information leakages with respect to micro-architectural events of the system such as cache misses and branch misses. These leakages are considered to be benign for the normal applications, but, if monitored precisely, they result in revealing sensitive information of the cryptographic algorithm. Cryptographic algorithms, in spite of being mathematically strong, can leak secret keys through such micro-architectural events since the implementations of such cryptographic algorithms leave their execution footprints on the shared system resources.

Public-key cryptographic algorithms are indispensable in our modern day life because of critical applications like: authentication, key exchange, digital signatures etc, which are required in almost every applications we use while we connect to the Internet. In the pioneering work in [1] it was first shown that the time to process different inputs can be used as a side-channel information to find the exponent bits of the secret keys for RSA, Diffie-Hellman, Digital Signature Standard (DSS) etc. While there have been several proposed countermeasures to prevent Simple Power Analysis (SPA), there exists more powerful attacks such as the differential attacks which proves the simple side-channel countermeasures to be ineffective. The most popular countermeasures against Differential Power attacks (DPA) in ECC have been proposed in [2,3], namely scalar blinding, scalar splitting and point blinding. All of these countermeasures either randomize the secret scalar or the base point of the curve. In this paper, we target all three of these countermeasures with an asynchronous granular sampling of branch misprediction event using the Hardware Performance counters (HPCs) and show to the best of our knowledge, for the first time that all three of these popular countermeasures are ineffective to thwart such attacks.

HPCs are a set of special-purpose registers storing the counts of hardware-related activities within the microprocessor. These counters contain information on the internal activities of the processor and hence can be utilized for both attacks and their countermeasures. In [4], these HPCs are exploited as side channels for time-based cache attacks. HPC L1 and L2 D-cache miss counters have been exploited as side-channels in [4] for performing cache attacks on symmetric-key algorithms, like AES as in [5]. While the paper shows that the HPCs can be used as potential source of leakage, the attacks were sensitive to noise introduced through loops, branches and also compiler optimizations to retain the tables. In [6], it was first established that branch misses from HPCs can reveal the secret key in RSA. The paper targets SPA countermeasures to justify the information leakage through the branch predictors. But the attack as in its true form is ineffective on countermeasures that blind the secret. In the paper, exponent and message blinding were referred to be an effective countermeasure of the attack.

Motivation This paper extends the scope of HPC based micro-architectural attack to DPA protected public key algorithms. The proposed work exploits the granular sampling of branch mispredictions using `perf ioctl` system calls. The granular sampling allows capturing of detailed traces, which in turn makes attack on blinded scalar multiplication practical. In the remainder, we explain and solve the various challenges raised by this work.

Contributions In our paper, we target the harder problem of attacking the DPA secure implementations such as scalar splitting and scalar blinding using the asynchronous branch misprediction samples. We use the `perf ioctl` system calls in sampling mode to monitor the executable under attack. *ioctl* (or input/output control) is a system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls. It takes a parameter specifying a request code and result in a device-specific response. When used with `perf_event_open`, it allows functionality like enabling, disabling, resetting, refreshing etc. of the counter. The sampling software samples number of branch misses suffered at a defined sampling period of some other sampler event (instruction count is used as the sampler event in our measurements). The granular samples obtained are noisy, mostly because the measurements are asynchronous in nature and thus noise is not uniform. This problem is difficult because of the absence of proper synchronous and granular measurements of mispredictions at every regular time-step. The algorithm being randomized in nature adds to the difficulty of attacking with such coarse measurements. We follow by a principle of *Deduce & Remove* in order to cope up with such limitations. We list the criteria of our attack on blinded Scalar multiplication as follows:

- *Acquire*: obtain branch misprediction traces over the scalar multiplication.
- *Deduce*: every randomized trace should reveal partial key bits.
- *Remove*: if a randomized trace does not leak any information regarding the trace, then the attacker should be able to isolate and remove the trace.

The objective of the *Deduce* step is to derive the values of an unknown set of keys based on the observed performance events. However, an important drawback for the success of iterative attack algorithm is that an erroneous deduction will affect subsequent key recoveries. Hence, the *Remove* step eliminates observations that have a low confidence or are erroneous by introducing extra checks.

The key contributions of this work are:

1. We perform a reverse engineering of the branch predictor hardware and found that the behavior has a significantly high correlation to the deterministic 3-bit predictor characteristics.
2. We propose a new method to perform side-channel attacks on the branch prediction unit, by building traces of branch mispredictions of any executable. We use hardware performance counters in sampling mode to build such traces, with user privileges.
3. We use these branch misprediction traces to construct templates, and later apply adaptive template matching to retrieve the secret scalar, including in presence of differential attack countermeasures.

2 Background

2.1 Understanding Branch Mispredictions

Commonly, the implementations of public-key exponentiation algorithms and the scalar multiplications algorithms in ECC are such that the sequence of operations executed in every run of the algorithm is dependent on the secret bits. Both the exponentiation and scalar multiplication algorithms are commonly implemented with a set of statements in `if-else` block and the execution of the if-else statements are conditionally dependent on the secret key bits. The relation between these conditional sequences and branch misses is the following. Let the n-bit secret scalar in ECC be denoted as $(k_0, k_1, \cdots, k_i, \cdots, k_{n-1})$. The double and add operations of the double-and-add algorithm or the SPA resistant Montgomery Ladder algorithm being conditioned on the secret scalar bits, the trace of taken or not-taken branches as conditioned on scalar bits and expressed as $(b_0, b_1, \cdots, b_{n-1})$.

- If a particular key bit k_j is 1 then the conditional addition statement in the double and add algorithm gets executed. Thus, the condition is checked first, and if the particular key bit is set then its immediate next statement ie, addition gets performed. Since this is a normal flow of execution the branch is considered as not-taken ie, $b_j = 0$ in this case.
- While when $k_j = 0$, the addition operation is skipped and the execution continues with the next squaring statement. Thus, in this case branch is taken ie, $b_j = 1$.

Thus for any `if-else` block, we consider the respective branch statement to be `not-taken` if the `if` conditional satisfies. On the other hand, if the `else` block is executed then we consider the respective branch to be `taken`.

The history of `taken` and `not-taken` branches are available to the branch predictor and the predictor predicts next branches based on the history of branches that have already been encountered. Whenever, the predictor encounters a conditional statement, it predicts based on the history and predicted instructions gets fetched in the instruction pipeline. It is only during the execute stage that the condition gets evaluated and if there is a mismatch in the predicted and the evaluated branch then the corresponding instruction is flushed from the instruction pipeline resulting in pipeline stall, which is commonly referred to as branch misses.

Effect of Compiler Optimization Options In order to validate our understanding for conditional branching we performed some experiments to observe the effect of optimization options in gcc on the conditional if-else structure of code. Similar to the balanced structure as in Montgomery ladder we show an example of assembly generation for a simple conditional if-else code. The code prints "hello" if the `if` clause is true otherwise it prints "hi".

```
.LC3:
.string "hello"
.LC4:
.string "hi"
```

So intuitively, if the "if" statement is true then the immediate next statement gets into the instruction pipeline and thus the branch is not taken. On the contrary if the "else" part is getting executed then then the branching is true and the branch statement is taken in such case. Table 1 shows the assembly translation of the code under various levels of optimizations. It is evident from all of these cases, that in spite of varying the optimization options, the conditional statement assembly remains the same.

2.2 Existing DPA Countermeasures on ECC

Elliptic curve scalar multiplication or point multiplication is an operation which computes $Q = K.P$, where K is n bit scalar and P is a point on the elliptic

Table 1. Assembly generated using various optimization options in gcc

without Optimization	O1	O2	O3
`.L5:`	`.L5:`	`.L3:`	`.L3:`
`    movl -36(%rbp), %eax`	`    cmpb $49, (%rsp,%rbx)`	`    movl $.LC4, %edi`	`    movl $.LC4, %edi`
`    cltq`	`    jne .L3`	`    call puts`	`    call puts`
`    movzbl -32(%rbp,%rax), %eax`	`    movl $.LC3, %edi`	`.L5:`	`.L5:`
`    cmpb $49, %al`	`    call puts`	`    .....`	`    ...`
`    jne .L3`	`    jmp .L4`	`    jne .L3`	`    jne .L3`
`    movl $.LC3, %edi`	`.L3:`	`    movl $.LC3, %edi`	`    movl $.LC3, %edi`
`    call puts`	`    movl $.LC4, %edi`	`    ....`	`    call puts`
`    jmp .L4`	`    call puts`		`    ...`
`.L3:`			
`    movl $.LC4, %edi`			
`    call puts`			

curve. The ECC scalar multiplication algorithms operate for each bit of the scalar and branching decision depends on the bit value of the scalar. Although scalar multiplications can be written without if-else blocks however many legacy codes still use them and it is also a belief that such if-else based codes with DPA protections can thwart these BPU based attacks.

Scalar Randomization In the paper [2], scalar randomization has been proposed as a countermeasure against Differential Power Attacks (DPA) for ECC. If K is the secret scalar and $P \in E$ the base point, instead of computing K times P, the paper suggests to randomize the scalar K as $K' = K + r * \#E$ where r is a random integer and $\#E$ is the number of points in the curve. The countermeasure computes $K'P$ which returns the same value as KP since $\#E.P = O$.

Scalar Splitting This countermeasure was proposed in [3] to randomize the scalar such that instead of computing KP, the scalar is split in two parts $K = (K - r) + r$ with a random r, and multiplication is computed such on the split components separately, $KP = (K - r)P + rP$.

Point Binding This countermeasure was also proposed in [2], and computes $K(P+R)$ instead of KP, where R a secret-random point and KR can be stored in the system beforehand, which when subtracted $K(P + R) - KR$ gives back KP.

2.3 Related Work on Branch Prediction Attacks

Asymmetric-key cipher implementations typically have key-dependent conditional branching statements which when implemented on systems with branch predictors, are subjected to side-channel attacks exploiting the deterministic branch predictor behavior due to their key-dependent input sequences. In [7] the penalty for mispredicted branches in number of clock cycles is observed as a side channel to identify the data-dependent operations of the public-key cryptographic system. On a standard RSA implementation, four different types of attacks were performed exploiting the Branch Prediction Unit (BPU) by using both synchronous and asynchronous techniques. Using timing as the side-channel in [7], the misprediction information is modeled to identify the secret key. In addition, the synchronous and asynchronous attacks involve the Branch Target

Buffer (BTB) to be modified by the attacker to surface the attack. A further improved version of this attack [8], [9] has also been carried out with proper knowledge of underlying hierarchical Branch Target Buffer (BTB) architecture of the target system.

The work in [7] has been extended by the authors in [6] using the Hardware Performance counters (HPCs) present in recent processors. The paper showed that the behavior of the hardware predictor bears a similarity to the deterministic 2-bit predictor state machine. Using this behavior, the authors targeted the unprotected or SPA resistant implementations of RSA and revealed the secret using branch misses from HPCs.

In [10], the authors introduced a new covert channel to perform secret communication between the sender and the receiver processes, by exploiting the residual state of dynamic branch predictor behavior of the system. While in [11], the authors described an implementation of Contention-based Covert channel. The authors also provided a comparison of both types of covert channel communication in both the single-threaded and Simultaneously Multi Threading (SMT) environment. An attack has been developed in [12] to derive kernel and user-level ASLR (Address Space Layout Randomization) offset which exploits the BTB collisions between the branch instructions. In [13], techniques for implementing binary exponentiation algorithms without requiring branch instructions have been proposed. However, the study of using HPCs to exploit the cipher codes implemented with branch statements is vital because there still exist several standard implementations using branches (as in OpenSSL [14], [15]). Though other implementations have been proposed without conditional branches, they have been subjected to side-channel attacks other than timing [16, 17]. HPCs, in particular, provide side-channel information to an adversary, which may be more powerful than timing information, and hence may be utilized to break implementations that prevent timing analysis.

2.4 Related Work on Micro-architectural Reverse-Engineering

There exists few papers have tackled the task of reverse engineering the branch predictor. In particular, [18] determined the size and organization of the BTB as well as the length of local and global branch history components, on Intel P6 and NetBurst architectures. Authors in [19], reverse-engineered more details of these structures, including interactions between different structures, focusing on the Pentium M architecture. The reverse engineering of branch predictors have been mostly considered from a compiler optimization perspective. They thus mostly focus on the *structures* of the branch predictor rather on the predictor itself, using microbenchmarks. Indeed, knowledge of the structures of the branch predictor is useful to optimize compilers and avoid destructive interference in tables [20]. As these works were published respectively on 2004 and 2009, they also target older processors. In the recent years there has been some attempts to reverse engineer the complex Last Level Cache (LLC) addressing scheme in [21] using the information from the HPCs. The DRAM addressing has been successfully reverse engineered in [22] using the timing channels.

3 Profiling the Branch Prediction Unit Using Asynchronous Measurements

In this paper, we change the perspective on the reverse engineering of branch predictors, having side channels in mind. Instead of relying on microbenchmarks, we use performance counters to create a trace of monitored mispredictions, and reverse the predictor according to the known branch traces fed as inputs. Compared to event-based sampling, we obtain fine-grained traces, allowing us to infer more information than with previous methods. We reverse engineer the dynamic branch predictor on the most recent Intel micro-architectures, namely Sandy Bridge and Broadwell.

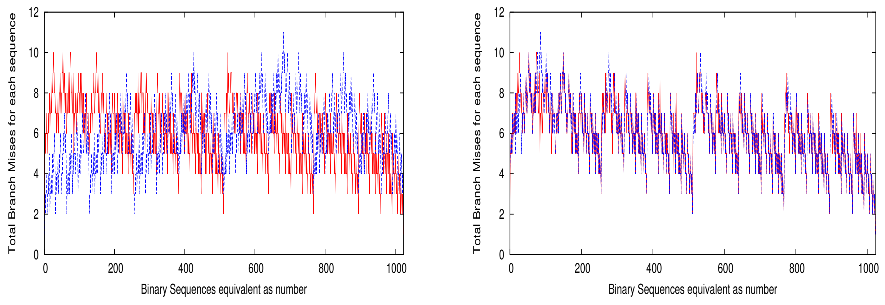
3.1 Reverse Engineering of Branch Predictors in Modern Intel Processors

In this paper, we make an initial attempt to understand the nature of the underlying branch prediction unit. The branch prediction hardware design is proprietary of the processor manufacturer and the precise details are not available. Thus, to properly exploit it for our attack, some reverse engineering of the hardware is required. In our experiments we use `perf_event_open` syscall across simple conditional statement execution to monitor branch misses from such set of instructions. We provide the code snippet of the `perf` class, which when instantiated with the required hardware event, returns the event count for the execution under consideration. So in order to understand the nature of the underlying branch prediction hardware we observe branch misses across a set of `if-else` statement block where the condition of execution of the `if` or an `else` statement depends on the input sequence.

```
static long
  perf_event_open(struct perf_event_attr *hw_event, pid_t pid,
                int cpu, int group_fd, unsigned long flags)
  {
     int ret;

     ret = syscall(__NR_perf_event_open, hw_event, pid, cpu,
                   group_fd, flags);
     return ret;
  }
   void start()
  {
    int rc = ioctl(fd_, PERF_EVENT_IOC_RESET, 0);
    assert(rc == 0);
    rc = ioctl(fd_, PERF_EVENT_IOC_ENABLE, 0);
    assert(rc == 0);
  }
  size_t stop()
  {
    int rc = ioctl(fd_, PERF_EVENT_IOC_DISABLE, 0);
    assert(rc == 0);
    size_t count;
    int got = read(fd_, &count, sizeof(count));
    assert(got == sizeof(count));
    return count;
  }
```

Using this `perf_event_open` system call, we can monitor the required hardware event across a code snippet from the user privilege. In our experiments,



(a) **Branch misses from 2-bit predictor and actual hardware predictor**

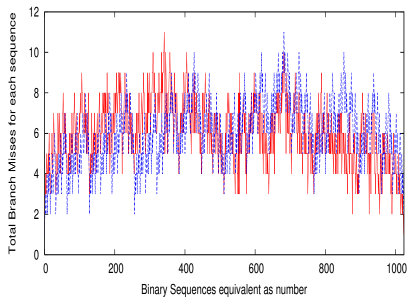
(b) **Branch misses from 3-bit predictor and actual hardware predictor**

Fig. 1. Branch misses from Intel i5-5200U with Broadwell Architecture

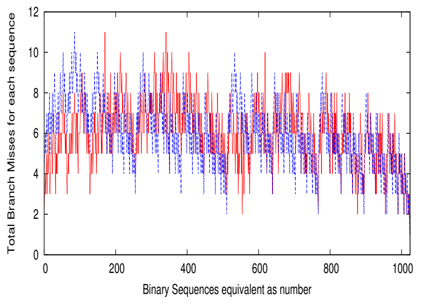
our code snippet has a conditional `if` statement conditioned on an input binary sequence. The `if` statement is triggered only when there is '1' in the sequence otherwise the `else` statement is executed. In order to understand how the hardware predictor works in our target system, we devised a simple experiment. We explain the experiments using the following steps.

- We have chosen binary sequences as input to the if-else conditional block of an arbitrary length 10 (wlog.). The experiments can be repeated for any sequence length. For our observation, we have all possible $2^{10} = 1024$ binary combinations of the input sequence as respective inputs to the code snippet.
- Branch mispredictions are observed as the performance monitoring event across each iteration of the if-else block on the input sequences.
- In the next phase, we simulated mispredictions over 2-bit predictor with the same set of input sequences and observed that there is a 65% match of individual mispredictions. In Figure 1(a), we plot aggregate mispredictions over each sequence for better illustration.
- Though in [6], the authors show that there is direct correlation of the overall branch misses reported for the 2-bit predictor to the perf-stat output, but in our experiments we observe nearly 65% accuracy for the granular observations of branch misses from 2-bit predictor and the actual hardware predictor.
- Further, we replaced the 2-bit predictor granular mispredictions with the 3-bit predictor simulated outputs. Figure 1 (b) shows the aggregate values of the mispredictions encountered for each sequence, while the individual accuracy of match is observed as 96%. Excluding some particular input sequences, the granular observation of branch misses from the hardware predictor using `perf_event_open` system matches exactly to the branch mispredictions simulated over 3-bit predictor.

Figure 2(a), 2(b) refer to the granular branch misprediction observations vs the behavior of simulated mispredictions from 2-bit and 3-bit predictor on Intel Sandybridge architecture. For all of the 4 figures, the `x-axis` shows the decimal equivalent of binary sequences and the `y-axis` shows the branch mispredictions for each such sequence. The red curve indicates the branch misses from the unknown hardware predictor for each binary sequence, while the blue curve indicates those observed over the deterministic 2-bit or 3-bit predictor. Clearly



(a) **Branch misses from 2-bit predictor and actual hardware predictor**



(b) **Branch misses from 3-bit predictor and actual hardware predictor**

Fig. 2. Branch misses from Intel i3-M350 with Sandybridge Architecture

from the figures it can be observed that there is a huge similarity in behavior of the underlying hardware predictor to the 3-bit predictor.

In the next section, we demonstrate that we do not require an elaborate reverse engineering of the branch prediction hardware in the system to reveal the secret scalar from the elliptic curve multiplication. We will follow by our observations of similarity of the behavior of the 3-bit predictor to the hardware predictor to retrieve secret even in presence of scalar blinding.

3.2 Sampling Granularly Using `perf ioctl` Calls

In the previous section, we discussed the granularly observing the branch misses over each iteration of the `if-else` block. Such measurements are not practical for an attacker, as the attacker cannot modify the executable run by the victim. Instead, the attacker can sample `perf` event counter values. The sampling mode of `perf ioctl` calls works in the following manner [23]:

- The following code snippet defines a signal handler, which defines the function to execute when there is an interrupt raised by the interrupt handler on overflow of the event.
- The idea is such that the `perf` object is instantiated with an event that is used as a sampler. For example, if `instruction count` event is used as the sampling event, there is a parameter `sample_period` which when set to a desired value, an interrupt is issued when the counter exceeds the `sample_period`.
- The interrupt handler is called at regular intervals of the `sample_period`. In our experiments, we measured the branch miss event counter on each such interrupt. This provides any user-level process a handle to granularly monitor any cryptographic module running in the system sampled at particular frequency of execution.

```
//Function to handle overflow interrupts
static void perf_event_handler(int signum, siginfo_t* info, void* ucontext) {
     if(info->si_code != POLL_HUP) { //Only POLL_HUP should happen
         exit(EXIT_FAILURE);
     }
     count++;
     read(fm, &sample_counter, sizeof(long long)); //Read branch_misses value
     ioctl(fm, PERF_EVENT_IOC_RESET, 0); //Reset the branch_misses counter to zero
     fprintf(fp, "%lld  ", sample_counter);
     ioctl(info->si_fd, PERF_EVENT_IOC_REFRESH, 1); //Refresh the instruction count buffer
}

     //Configure the signal handler
struct sigaction sa;
     memset(&sa, 0, sizeof(struct sigaction));
     sa.sa_sigaction = perf_event_handler; //Specify what function to execute on interrupt
     sa.sa_flags = SA_SIGINFO;
```

Though the counter values read by the interrupt handler for a very small `sample_period` is observed to be more noisy, due to the overhead of the interrupt getting generated very frequently. The optimal sample period can be decided on the target machine testing with some dummy process. In our experiment platform, we found that the sampling becomes more noisy if we sample with sampling period less than 50 instructions.

3.3 Threat Model

In the previous section, we show `perf ioctl` call can be used from the user space to monitor hardware events associated with some executable. The measurements made in our setup are such that there is an event (branch misses) that is getting monitored on specific sample period of the instruction count. This measurement is purely handled by the `ioctl` interface and is observed to be asynchronous in nature. In the following section, we develop an attack algorithm which monitors the branch misprediction traces using this asynchronous `perf` output and retrieves the secret.

The attack model is both practical and realistic in virtualized shared environment where the hardware is typically shared between multiple users processes. In such setting, the branch misprediction event counts can thus be observed over a target execution by the attacker from another concurrently running process. The hardware being shared, the mispredictions from one execution has an effect on the concurrent running process as well, making our attack model realistic.

4 Attack Overview

In this section, we develop a general attack methodology which can take perf event samples over the period of execution and can reveal the secret using simple statistical techniques such as template building and template matching. In this paper, we follow by a strategy of *Deduce & Remove* to target the scalar splitting and scalar blinding countermeasures which randomizes the scalar multiplications. In most recent cryptographic libraries, the underlying scalar multiplication algorithm is balanced and the scalar is blinded using a newly generated random value every time.

Before going into the actual description of the attack strategy, let us define the parameters. Initially we assume that there is an unprotected algorithm of ECC where the conditional execution is dependent on the scalar K. The considered parameters are:

- n-bit scalar K.
- m branch miss samples from the execution over K.
- Each branch miss sample is reported after sample period of I instructions.

Algorithm 1: **Template Building**

```
Input: n: number of bits of scalar $K$, m: number of branch miss samples for $n$ bits scalar.
       Acquired samples of $2^t$ sequences iterated over $i$ times for each of the $2^b$ predictor
       states
Output: Templates corresponding to $2^t$ sequences of $t * m/n$ sample points for each $2^b$ state
begin
    for $2^b$ states of the b-bit predictor do
        for each of the $2^t$ sequences do
            Construct separate distributions of $i$ values for each $t * m/n$ sample points.
            Compute separately, the mode of each distribution to be the elected template
            for that sample point.
        end
    end
end
```

- Thus effectively, each sample of reported branch misses is affected by n/m bits of the scalar K.
- In our experiments, we have chosen I such that $n/m = 2$. Moreover, considering a b-bit predictor, I should be such that $n/m \leq b$ ($b = 3$ for our case).

4.1 Offline Template Building for Each State of *b*-bit Predictor

We propose the template building phase with the context of states of a generic b bit predictor. There are 2^b states of the b-bit predictor and each of them refer to a particular sequence of the last taken and not-taken branches. If we encode taken branches as 1 and not-taken branches as 0, then each state represents the history of the last consecutive branches, and the future branches will lead to mispredictions following the behavior of the current state. In simple words, a given sequence of taken and not-taken branches may exhibit different number of mispredictions depending on the state of the predictor at the start of the sequence. Since the nature of branch misprediction for the same sequence is different for different start states, this motivates us to build templates for the same set of input sequences for each of 2^b states of the b-bit predictor.

- Branch miss templates from scalar multiplication over t bits ($t << n$) are constructed in sampling mode.
- Each of 2^t template contains $t * m/n$ sample points.

At the end of the template building stage as in Algorithm 1, we obtain 2^b set of templates, each having 2^t templates of $t * m/n$ sample points.

4.2 Offline Template Matching for an Unknown Trace

Template building phase is followed by a matching phase, where sample trace collected for an unknown secret scalar is matched iteratively to the previously constructed templates. The matching phase is composed of: *Deduce* and *Remove* steps. In *Deduce* phase, we start matching from the Least Significant Bit (LSB) of the scalar multiplication. We assume here that the adversary has the knowledge of the start state of the execution. This assumption is indeed practical since the adversary can flush the branch predictor hardware configuration by execution a long trail of all taken or all not-taken to put the predictor hardware in one of the extreme states and allow the victim process to follow his own execution. Since the hardware predictor is shared by all the processors in the system, this makes the template matching easier in the next step.

The matching can be done iteratively taking on a trace with s sample points ($s = t * m/n$). These s samples are point-wise matched with all the template points for each particular template and the distances for each of the traces are measured using the Least Square Method (LSM). A set of templates having the least squared distance to the unknown template is considered as the retrieved t bits of the unknown scalar. In a noise-free setting, a single trace matching should be sufficient to determine t-bit scalar. However, in a real setting where noise is predominant, several templates might return same least square distance. The noise filtering is done in *Remove* step. However, we noticed that *Remove* is device-specific and can also change with algorithm. Thus is discussed later for our implementation.

At the end of the *Deduce* and *Remove* steps, the retrieved t bits decides the intermediate state of the branch predictor hardware. With the current t bits, the rest of the scalar is determined with the same procedure iteratively, extracting t bits at a time.

5 Attacking Exponent Splitting

The idea of exponent splitting as a DPA countermeasure for scalar multiplication appears in [3, 24]. Authors in [3] proposed a method to randomize the scalar K such that instead of computing KP, we can compute $(K - r)P + rP$ where random r changes on every run. In this paper, we are going to show that, such secure implementation is still vulnerable to branch misprediction analysis attacks. We assume that SPA resistant scalar multiplication are computed on each of the splits separately, over a balanced scalar multiplication algorithm. The SPA resistant scalar multiplication on split shares together result in DPA resistant algorithm.

5.1 Adapting the Generic Template Matching to Scalar Splitting

The attack progresses from LSB to MSB. In the following discussion, we explain the iterative attack algorithm to recover the bits in the secret scalar K, starting from LSB. Templates composed of trace pair corresponding to all possible values of $(K-r)P$ and rP are constructed. The attack algorithm works in three major steps:

1. *Acquire*: N pairs of split scalar multiplications over $K-r$ and r are acquired over t bits, each pair for unknown and random values of r.
2. *Deduce*: For each of the N pairs, corresponding pairwise template matching is performed, on each sample. It results in N values each for $K - r$ and r. Pairwise adding up of each pair $(K - r + r)$ results in t-bits of K.
3. *Remove*: Ideally, all N values of K obtained previously must be identical. The non-matching values can be removed by majority voting.

The algorithm is thus repeated iteratively, revealing t bits at a time.

6 Attacking Scalar Blinding

In this section, the attack is extended to scalar blinding countermeasure. Although the generic attack principle stays the same, minor tweaks are needed when applied to blinding. In scalar blinding, the algorithm computes $K' = K + r\#E$, where the secret scalar K is not operated upon by the scalar multiplication algorithm, instead the scalar is randomized every time with a new random number r and is multiplied with the order of the curve $\#E$. There are no further adjustments that needs to be done after scalar multiplication of $K'P$ since the algorithm results in the exact same value as KP in spite of varying r for every run.

6.1 Adapting Generic Template Matching for Scalar Blinding

The attack algorithm has been modified to tackle the traces from scalar blinding. Unlike scalar splitting, the scalar cannot be retrieved iteratively, as the modulus operation on each of the blinded scalars can be applied only on full scalar. The final value retrieved after modulus can be checked for each of the N samples to remove the incorrectly retrieved candidates.

The offline template building phase is same as the previous attack.

1. *Acquire*: N blinded scalar multiplication over $(K + r\#E)P$, for random, unknown values of r.
2. *Deduce*: For each of the N trace, pointwise matching over s branch misprediction samples of t bits is performed. It results N candidates for t bits for of $K + r\#E$.
3. Remove: Choose any 3 branch misprediction traces out of N traces, for random r_1, r_2, r_3. Step 2 reveals t bits of $K+r_1\#E$, $K+r_2\#E$ and $K+r_3\#E$ respectively. Take pair wise difference of the candidate values example $(K+r_1\#E)-(K+r_2\#E)$. Compute $r_1\#E-r_2\#E$, $r_2\#E-r_3\#E$ and $r_1\#E-r_3\#E$. Now for correct t bits of the blinded scalar, adding up of candidate value of $r_1\#E-r_2\#E$ and $r_2\#E-r_3\#E$ would result in non-empty set on intersection with candidate of $r_1\#E - r_3\#E$. Combination for empty set for intersection can be discarded, leading to t bits of blinded scalar.

This algorithm is thus extended for the next s samples with adaptively performing template matching and analysis based on the knowledge of the retrieved t bits. One final *check* after retrieving the entire length of the blinded scalar is to perform a modulus operation on the retrieved blinded components with the order of the curve. All the independent retrieval of the n bits should produce same secret scalar K on taking the modulus.

7 Experimental Results

In this section we will illustrate each steps explained in the prior sections with results from performance counters. We will start with the acquisition of traces. The sample points of the trace of branch misses obtained from sampling ioctl calls are noisy and are highly sensitive to the sample_period of the sampling event. The sampling being asynchronous to the underlying execution, we acquired several traces of same input sequence in order to construct the template. The parameters that we chose for template building and matching is tabulated in Table 2. Having set the `sampling_period` as 80, results in having one sample point affected by two bits of the input scalar.

Table 2. Tabular Representation of Symbols

Symbols	Values
Number of sample points for template matching (s)	5
Number of bits considered for template matching (t)	10
Sampling period (instructions)	80
Number of independent traces collected (N)	10000

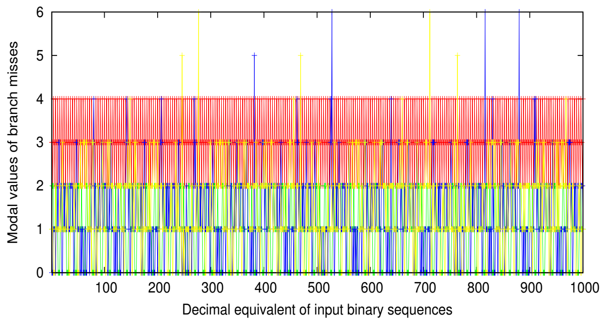


Fig. 3. Variation of sample points of branch miss templates

7.1 Building Branch Misprediction Templates Using perf ioctl Sampling

The success of our attack is highly dependent on how accurate the template has been built. Choosing these 5 sample points corresponding to a particular predictor state, is the most difficult task. The sample of branch misses obtained over the input sequences being noisy, we repeat the execution over the same sequence multiple times to remove the effect of the noise while building the template.

Initially, we constructed the templates considering the mean of the obtained samples. The templates as constructed using the mean value loose the correlation to the behavior of the 3-bit predictor since the sample means get affected by the sampling noise. Taking the sample mean is the most preferred technique for Correlation Power Attacks (CPA), but the nature of the sample noise is irregular or uneven in this case, for this reason mode of the distribution outperforms mean. We separately construct frequency distributions for each of these sample points and select the modal value (the sample value which has occurred the most) as the candidate template point.

As in the Figure 3, we show the variation of the respectively selected modal values for all possible of 10 bit sequences. The graph is plotted with the decimal equivalent of the binary number in the x-axis and the behavior of the modal sample value with the input binary sequences in the y-axis. Each colored line in the figure represents a sample point (affected by two bits), and if they are studied keenly represents a high similarity to the behavior of the 3-bit predictor. Thus we claim we that the templates constructed taking the highest frequency points capture the essence of the distribution accurately.

7.2 Retrieving the Least Significant Bit for Scalar Splitting

Among the observed sample points, the most noisy sample point is the first one, which is supposed to be affected only by the Least Significant bit, and the bit

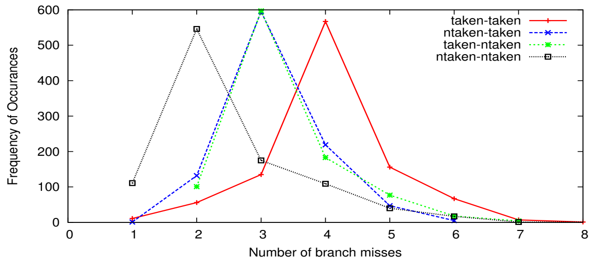


Fig. 4. Confusion in determining the LSB for scalar splitting

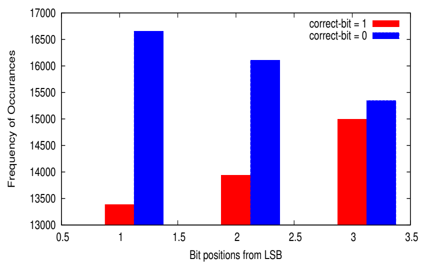
following the LSB and but often also gets affected by the branch misses from instruction before the scalar multiplication starts to execute. The sampling being asynchronous to the underlying execution, this sampling noise has to be handled intelligently in order to diminish the chances of error.

In Figure 4, we have performed a frequency analysis on the first branch miss samples observed over a set of random binary sequences of input. Depending on the actual values of the LSB and the bit following LSB we separately constructed the frequency distribution of the observed samples. At the start of this execution, the hardware branch predictor state can be assumed to be in `strong not-taken` state, since we execute a long trail of `not-taken` branches before the start of execution. The 3 bit predictor being a 3 bit saturating counter, a long trail of `not-taken` branches puts the predictor state to one of the two extreme states of the saturation counter.

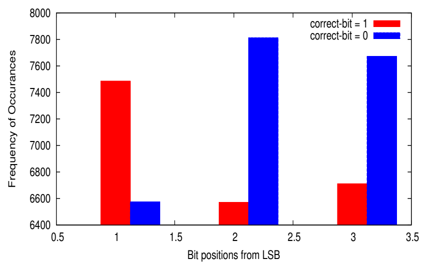
As appears in Figure 4, when the last two bits of the sequence is having both `not-taken` branches then the distribution is shifted towards the left (in black line) and exhibits overall lesser branch misses from the rest of the three cases. While on the other hand if the sequence is having both `taken` branches (in red line) is suffering from the most branch misses and the distribution is shifted towards the right. While the other two cases (in green and blue line) of combination of one `taken` and `not-taken` cannot be distinguished from each other by simple template matching.

Thus to retrieve the LSB separately, we take the N pair of samples for each of the split scalar and the random component such that if they are having sample values in range $(3, 4)$ we are not going to classify them. This is because for each of the four distributions there is a large frequency of values in this range, thus encountering a sample as 3 or 4 will add up to the confusion. In the following steps we explain the working procedure to retrieve the LSB

– For each of the N sample pairs, we select the pairs where neither of the sample points exhibits a value 3 or 4.
– If a sample point exhibits value < 2, we classify both the branches as `not-taken` ie, the bits to be 11.
– If a sample point exhibits value 2, we conclude that the branches are either both `not-taken` or `not-taken` followed by a `taken` branch. Thus the bit values are 11 or 01.
– If a sample point exhibits value 5, we conclude that the branches are either both `taken` or `taken` followed by a `not-taken` branch. Thus the bit values are 00 or 10.



(a) **Determining next three bits**



(b) **Determining further three bits**

Fig. 5. Determining 3 bits at a time for scalar splitting

– If a sample point exhibits value > 5, we classify both the branches as `taken` ie, the bits to be 00.

After we conclude the last two bits from M pairs out of the N, we take their pair wise sum to conclude the LSB of the secret scalar K. One additional checkpoint we have with this approach is that we can check that whether each M pairs leads to the same LSB of the secret scalar K.

7.3 Iterative Template matching for scalar splitting

In this section we will put how to iteratively retrieve the further scalar bits applying the *Deduce* and *Remove* strategy once the LSB is known. The first step is that we perform template matching on first $t = 10$ bits and deduce the candidate values which match respectively to each of the N pair of samples. Next we take pair wise summation of such values, and that gives us the candidate values of the first t bits of the secret K. At this point, we remove the candidates which wrongly infer the LSB. Figure 5(a) show the majority voting of the next three bit positions from the LSB for the candidate values, where clearly the correct bits (three consecutive one's) wins with a significantly high majority. One striking observation is that, this winning percentage gradually reduces for the further bits. Behavior of the branch misses having huge similarity to 3 bit predictor characteristics, thus we decide only on the next 3 bits. Once we identify three bits, we retrace the split scalar values individually and remove the candidate values for both $K - R$ and R which do not add up to retrieved bits.

Thus, with the knowledge of the LSB we perform windowed template matching on the t next bits by sliding the window every time with the knowledge of 3 bits. Figure 5(b) reveals the next 3 bits as in $(0, 1, 1)$ based on adaptive template matching (ie, matching the templates after knowing the intermediate state of the predictor for the retrieved bits). For the further bits, we iteratively apply the same adaptive template matching to reveal the bits without an error.

7.4 Efficiency of Deduce and Remove strategy on Scalar Blinding

As discussed in the section 6, an attacker uses *Deduce* and *Remove* strategy on all of the acquired N branch misprediction traces. Performing the easy check as appears in section 6 can significantly remove the traces that are too noisy

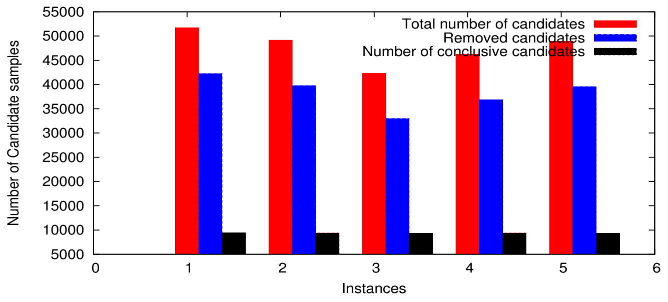


Fig. 6. Efficiency of Deduce and Remove step on scalar blinding

to leak any useful information. This brings us to analyze the effectiveness of the two strategies. The online trace acquisition and template building phase works exactly same as has been explained for the scalar splitting algorithm. In the following discussion, we will explain the analysis of the template matching phase and pruning out the wrong predictions.

The attack algorithm in section 6 is such that the secret scalar can be retrieved only when the full length of the blinded scalar has been retrieved entirely. There is no iterative check so as to detect if there is error. In order to detect we discussed a check mechanism in the attack section which helps to correctly retrieve first 3 bits of each random scalar and this time we update the predictor states after each 3 bits and proceed for retrieving the subsequent bits. The template matching phase in scalar blinding works on N misprediction traces, in our case we choose $N = 10000$. For each of these blinded scalars after performing template matching by the Least Squared method, they may have multiple candidate values to match the traces. The reason is simply because the trace acquired is noisy. Each blinded trace being random in nature, we denote the matched candidates for each trace as N sets: $R_1, R_2, \cdots, R_N$, where each set is represented as $R_i = \{c_j : j \geq 1, c_j$'s are candidate matched templates of t bits$\}$

- We consecutively take three sets from the entire range. It is not possible to exhaust all $\binom{N}{3}$ combinations, so we choose in such a fashion.
- Thus we choose sets like R_i, R_{i+1}, R_{i+2} and performed subtractions on all pairs of the candidate values as explained in section 6.

The efficiency of this Deduce and Remove strategy is illustrated in Figure 6, where the bars with red represent the total number of candidates which were matched after template matching. The bar with blue indicates the total number of candidate values that were pruned. The last bar in black represents the number of correctly retrieved candidates after taking intersection. The number of correctly retrieved ones are higher than 93%. That is out of 10000 separate random traces, more than 93% of the traces could identify the last 3 bits correctly among the various candidate values. Knowing 3 bits we update the state of the predictor and perform template matching on the next t bits to retrieve the following 3 bits.

7.5 Adapting Generic Template Matching for Point Blinding

The attack algorithm as explained for scalar splitting and scalar blinding can be easily adapted for point blinding countermeasure. The countermeasure performs

$K(P+R)$ instead of KP. Since the number of branch mispredictions are not directly dependent on the point with which multiplication is getting done, so the conditional branching will be affected only because of the secret scalar K. Attacking the previous approaches make this attack a trivial one. The branch misprediction traces over the scalar multiplication $K(P+R)$ will always report branch misses for the scalar K. Thus the template matching phase will return candidate values for t bits of the scalar K every time. Considering that the traces obtained are noisy, we suggest to perform this on multiple traces and take an intersection between the candidate samples to eradicate the effect of noise.

Similarly, revealing t bits of the scalar leads the branch predictor to update the intermediate state after t bits, such that precise template matching can be applied on the subsequent t bits. This iterative algorithm is anticipated to require much less number of traces than the previous attacks.

8 Possible Countermeasures

Blinding countermeasures are shown to be vulnerable to branch misprediction attack in the previous sections. To counteract it, one can adopt several strategies. The first and the most obvious countermeasure is to implement the scalar multiplications such that the control flow of the execution is independent of the secret scalar K, thus avoiding the if-else structure of the implementation. But still there exists legacy codes which believe that the DPA countermeasures are sufficient to thwart branch misprediction attacks.

Another countermeasure which can thwart such attacks is to randomize the state of the predictor intermediate to the execution. This can be achieved by introducing random branching executions parallel to the execution of the cryptographic library. The objective is to randomize effectively the state of the predictor to inhibit the *Deduce* step of the attack.

Randomization in several layers of algorithm and measurement can only make the attack difficult in its original form. The adversary in some way can be more powerful having more granular traces and even then these countermeasures will pose themselves ineffective. This brings us again to the open challenge of rethinking the structure of the branch predictor in such a way that they are inherently secure against this form of attacks.

9 Conclusion

Information leakages from branch predictor pose a serious threat to the asymmetric key cryptographic algorithm because of their conditional statement execution depending on the secret. In the paper, we initially perform reverse engineering on the branch predictor hardware of Intel's Broadwell and Sandybridge systems and show that the hardware has a significant similarity in behavior to the 3-bit predictor algorithm. Subsequently, we use this granular observation of branch misprediction to attack the DPA secure implementations of ECC. The experimental results illustrate the effectiveness and efficiency of our proposed attack for both scalar splitting and scalar blinding countermeasures. Finally, we propose to execute random branches concurrently to randomize the hardware predictor state as a countermeasure against this attack.

References

1. Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, London, UK, 1996. Springer-Verlag.
2. Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, pages 292–302, 1999.
3. Christophe Clavier and Marc Joye. Universal exponentiation algorithm. In *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, number Generators, pages 300–308, 2001.
4. Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *FDTC*, pages 59–67. IEEE Computer Society, 2008.
5. Joseph Bonneau and Ilya Mironov. Cache-Collision Timing Attacks Against AES. In Louis Goubin and Mitsuru Matsui, editors, *CHES*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2006.
6. Sarani Bhattacharya and Debdeep Mukhopadhyay. Who watches the watchmen?: Utilizing performance monitors for compromising keys of RSA on intel platforms. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 248–266. Springer, 2015.
7. Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys Via Branch Prediction. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2007.
8. Onur Aciiçmez, Shay Gueron, and Jean-Pierre Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In Steven D. Galbraith, editor, *IMA Int. Conf.*, volume 4887 of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2007.
9. Onur Aciiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. Micro-Architectural Cryptanalysis. *IEEE Security & Privacy*, 5(4):62–64, 2007.
10. Dmitry Evtyushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. Covert channels through branch predictors: a feasibility study. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2015, Portland, OR, USA, June 14, 2015*, pages 5:1–5:8, 2015.
11. Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *TACO*, 13(1):10:1–10:23, 2016.
12. Dmitry Evtyushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. Jump over ASLR: attacking branch predictors to bypass ASLR. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 1–13, 2016.
13. David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side

channel attacks. In *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, pages 156–168, 2005.

14. Yuval Yarom and Naomi Benger. Recovering openssl ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.
15. 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.
16. Pierre-Alain Fouque and Frédéric Valette. The doubling attack - *Why Upwards Is Better than Downwards.* In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, pages 269–280, 2003.
17. Abhishek Chakraborty, Sarani Bhattacharya, Tanu Hari Dixit, Chester Rebeiro, and Debdeep Mukhopadhyay. Template attack on SPA and FA resistant implementation of montgomery ladder. *IET Information Security*, 10(5):245–251, 2016.
18. Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. Microbenchmarks for determining branch predictor organization. *Software: Practice and Experience*, 34(5):465–487, 2004.
19. Vladimir Uzelac and Aleksandar Milenković. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*, pages 207–217, 2009.
20. Daniel A Jiménez. Code Placement for Improving Dynamic Branch Prediction Accuracy. *ACM SIGPLAN Notices*, 40(6):107–116, 2005.
21. Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In Herbert Bos, Fabian Monrose, and Gregory Blanc, editors, *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.
22. Peter Pessl, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Reverse engineering intel DRAM addressing and exploitation. *CoRR*, abs/1511.08756, 2015.
23. Ubuntu Manuals. perf_event_open-set up performance monitoring, http://manpages.ubuntu.com/manpages/wily/man2/perf_event_open.2.html, 2017.
24. Jean-Luc Danger, Sylvain Guilley, Philippe Hoogvorst, Cédric Murdica, and David Naccache. Improving the big mac attack on elliptic curve cryptography. In *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, pages 374–386, 2016.