Branch Prediction Attack on Blinded Scalar Multiplication

Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin and Debdeep Mukhopadhyay

Abstract—In recent years, performance counters have been used as a side channel source to monitor branch mispredictions, in order to attack cryptographic algorithms. However, the literature considers blinding techniques as effective countermeasures against such attacks. In this work, we present the first template attack on the branch predictor. We target blinded scalar multiplications with a side-channel attack that uses branch misprediction traces. Since an accurate model of the branch predictor is a crucial element of our attack, we first reverse-engineer the branch predictor. Our attack proceeds with a first online acquisition step, followed by an offline template attack with a template building phase and a template matching phase. During the template matching phase, we use a strategy we call *Deduce* & *Remove*, to first infer the candidate values from templates based on a model of the branch predictor, and subsequently eliminate erroneous observations. This last step uses the properties of the target blinding technique to remove wrong guesses and thus naturally provides error correction in key retrieval. In the later part of the paper, we demonstrate a template attack on Curve1174 where the double-and-add always algorithm implementation is free from conditional branching on the secret scalar. In that case, we target the data-dependent branching based on the modular reduction operations of long integer multiplications. Such implementations still exist in open source software and can be vulnerable, even if top level safeguards like blinding are used. We provide experimental results on scalar splitting, scalar randomization, and point blinding to show that the secret scalar can be correctly recovered with high confidence. Finally, we conclude with recommendations on countermeasures to thwart such attacks.

Index Terms—Branch Prediction Unit, Scalar Multiplication, Scalar Splitting, Scalar Randomization, Point Blinding

1 Introduction

Micro-architectural side-channel attacks have gained importance manifold in the last decade [1]. These attacks target information leakage with respect to micro-architectural components such as the cache or the branch predictor. Cryptographic algorithms, in spite of being mathematically strong, can leak secret keys through micro-architectural events since their implementations leave execution footprints on the shared system resources.

Public-key cryptographic algorithms are indispensable in daily life due to authentication, key exchange, and digital signatures, which are required in almost every application we use while connecting to the Internet. In his pioneering work, Kocher [2] showed 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. Bhattacharya and Mukhopadhyay [3] first established that branch misses from Hardware Performance Counters (HPCs) can reveal the secret key in RSA. While there have been several proposed countermeasures to prevent Simple Power Analysis (SPA), there exist more powerful attacks such as Differential Power Analysis (DPA), proving simple side-channel countermeasures to be ineffective. In particular, while the authors [3] targeted SPA countermeasures, their attack is ineffective on countermeasures blinding the secret. The paper in [3] targets unblinded implementation of RSA and uses the branch misprediction values from perf over various inputs to construct an efficient Difference of Mean (DoM) based approach to identify the secret bits one after another. We target a difficult version of the problem in this paper, by focusing on the blinded implementations of different ECC algorithms because the blinded scalar multiplications (or exponentiation in case of RSA) are more commonly used in practice compared to unblinded operations. The Difference of Mean analysis based on the secret bits of the RSA exponents cannot be equivalently adapted for the blinded scalar multiplication with the aggregate branch misprediction values as described in [3].

The most popular countermeasures against DPA in Elliptic Curve Cryptography (ECC) have been proposed by Coron [4] and Clavier and Joye [5], namely scalar splitting, scalar randomization, and point blinding. All of these blinding countermeasures either randomize the secret scalar or the base point of the curve. They were considered to be effective countermeasures against HPCs side-channel attacks [3]. Such countermeasures may be observed in conjunction with many open source libraries like RELIC [6] which are used for development of cryptographic software and are promoted via corresponding literature [7].

Contributions

In this paper, we target all of these countermeasures, therefore tackling the harder problem of attacking DPA secure implementations. To the best of our knowledge, we show for the first time that all of these popular countermeasures are ineffective to thwart such attacks, contrary to what was previously believed. Our attack proceeds with the following steps: (1) an online acquisition phase, and (2) an offline template attack.

- *S. Bhattacharya is with COSIC, KU Leuven*
- *D. Mukhopadhyay is with SEAL, IIT Kharagpur, India*
- *C. Maurice is with Univ Rennes, CNRS, IRISA, France.*
- *S. Bhasin is with Temasek Laboratories, NTU, Singapore.*

The *online acquisition phase* provides the branch misprediction trace over the scalar multiplication. This trace is obtained with HPCs recording branch misprediction events, using the `perf_event_open` system call in sampling mode to monitor the executable under attack. This system call is part of the Linux kernel and accessible via ioctl. Contrary to previous work [3], which used HPCs in counting mode thus providing the aggregate number of occurring events, we use the sampling mode. The sampling mode provides periodic measurements with respect to another sampler event (e.g., instruction count). However, the obtained samples have a non-uniform noise, mostly because the measurements are asynchronous in nature, and the sampling period is not a fixed time period. The algorithm being randomized, adds to the difficulty of attacking with such measurements.

The *offline template attack* retrieves the secret key. The template attack is composed of a template building phase, followed by a template matching phase. During the template matching phase, we perform a *Deduce* step to derive the values of an unknown set of keys based on the observed performance events, and a *Remove* step to eliminate observations that have low confidence or erroneous by introducing extra checks. An erroneous deduction may affect subsequent key recovery, therefore the *Remove* step, as well as an accurate model of the branch predictor, are of crucial importance.

The key contributions of this work are:

1) We perform a reverse engineering of the branch predictor hardware and show that its behavior can be modeled by a 3-bit saturating counter state machine.
2) We propose a new method to perform side-channel attacks using the Branch Prediction Unit (BPU), by building traces of branch mispredictions of any executable. Contrary to prior works, to build such traces we use HPCs in sampling mode.
3) We use these branch misprediction traces to construct the first template attack on the branch predictor. Our adaptive template matching can be demonstrated on any secret-dependent conditional branching algorithm and is able to retrieve the secret scalar, including in presence of differential attack countermeasures.
4) We also demonstrate this attack on Curve1174 and Curve25519, implemented using the double-and-add always algorithm where the control flow of execution *does not directly depend on secret bits*.

Further we extend our attack to the open source implementations of Curve25519 and Curve1174 in RELIC. The attack strives on template matching algorithm which deduces the random scalar bits by using the constructed traces in a single match. The *Remove* algorithm is dependent on the countermeasure under consideration and can efficiently detect if a particular retrieval was wrong. Thus if the *Deduce* step fails, then the template matching must have returned wrong candidates for the current bit retrieval in majority of cases. This affects the retrieval of next bits as we need to increase the number of samples for a successful attack. In this paper, we present our experiments based on the commercially available Intel CPUs.

2 Background

2.1 Branch Predictors and Branch Mispredictions

Commonly, the implementations of public-key exponentiation algorithms and the scalar multiplications algorithms in ECC contain a conditional block, where the branches are conditionally dependent on secret key bits. This is the case in the double-and-add algorithm or the SPA resistant Montgomery Ladder algorithm. Let the n-bit secret scalar in ECC be denoted as $(k_0, k_1, \cdots, k_i, \cdots, k_{n-1})$. The trace of taken and not-taken branches depending on scalar bits is expressed as $(b_0, b_1, \cdots, b_{n-1})$.

- If $k_j = 1$, then the conditional addition statement in the double-and-add algorithm gets executed. The branch is not-taken, *i.e.*, $b_j = 0$.
- If $k_j = 0$, then the addition operation is skipped and the execution continues with the next squaring statement. The branch is taken, *i.e.*, $b_j = 1$.

In order to avoid pipeline stalls, the predictor predicts whether the next branch is taken or not, based on the history of branches that have already been encountered. Predicted instructions are then fetched in the instruction pipeline. The condition is only evaluated during the "execute" stage. In the case of a mismatch between the predicted and the evaluated branch, the corresponding instruction is flushed from the instruction pipeline resulting in a pipeline stall. This is called a *branch misprediction*, or a branch miss.

2.2 Existing DPA Countermeasures on ECC

Elliptic curve scalar multiplication or point multiplication is an operation which computes $Q = KP$, with K an n bit scalar and P a point on an elliptic curve. The ECC scalar multiplication algorithms operate for each bit of the scalar and the branching decision depends on the bit value of the scalar. Although scalar multiplications can be written without secret-dependent blocks, many open-source codes still use them. We came across 5 popular libraries namely: OpenSSL [8], libgcrypt [9], Bouncy Castle [10], mbedTLS [11], RELIC [6] which uses conditional structures in their codes. Some of these libraries have been already shown vulnerable by various attacks on the conditional structure of the codes [1], [2], [3], [12], [13]. Though the revised version of some of these libraries are sanitized with the recent developments of micro-architectural attacks, there are still inherent data-dependent conditional blocks of codes which eventually depends on the cryptographic secret.

Moreover, it is believed that BPU-based attacks can be thwarted by DPA protections *even with secret-dependent code*. The countermeasures are scalar splitting, scalar randomization, point blinding and randomized projective coordinates.

Scalar Splitting

Clavier and Joye [5] proposed, instead of computing KP, to split the scalar in two parts $K = (K - r) + r$ with a random r. Multiplication is then computed on the split components separately, *i.e.*, $KP = (K - r)P + rP$.

Scalar Randomization

Coron [4] proposed randomization of the scalar such that for K the secret scalar and $P \in E$ base point, instead of computing KP, we randomize K as $K' = K + r \times \#E$, with r a random integer and $\#E$ the number of points in the curve. The countermeasure computes $K'P$ which returns the same value as KP since $\#E.P = O$.

Point Blinding

Coron [4] proposed point blinding, *i.e.*, to compute $K(P + R)$ instead of KP, where R is a secret-random point. KR can be stored in the system beforehand, which when subtracted $K(P + R) - KR$ gives back KP.

Randomized Projective Coordinate

This countermeasure proposed by Coron [4] involves a randomization of the projective coordinate representation $P(X, Y, Z)$ of point $P(x, y)$ by a randomly selected λ in the finite field. The transformation $P(X, Y, Z)$ to $P(\lambda{\cdot}X, \lambda{\cdot}Y, \lambda{\cdot}Z)$ can be applied once for the entire scalar multiplication or alternatively after each individual addition doubling steps.

3 Reverse-Engineering Branch Predictors

In order to exploit the state of the branch predictor, we need an accurate model of it. However, the hardware design of this proprietary component is often undocumented. Previous work [3] assumed that the predictor is a 2-bit saturating counter. In this section, we seek to evaluate how well this model corresponds to the actual branch predictor. We focus on Intel CPUs and provide a detailed comparison among various micro-architectures.

3.1 Reverse-Engineering Method

Branch predictors are complex structures comprising tables associated with different history lengths. As we are focused on attacks on cryptographic algorithms which usually target a single branch, we do not seek to reverse-engineer these components, but rather the prediction method. We thus assume that the component structure has a negligible impact on our attack.

We model the predictor as a state machine. The different states map to either a “branch taken” or “branch not taken” prediction for the next branch. The state machine is updated with each new branch, the input being either “branch taken” or ”branch not taken”. As an illustration, a 2-bit saturating counter has four states: “strongly not taken”, “weakly not taken”, “weakly taken” and “strongly taken”, where the first two states map to a “branch not taken” prediction and the last two a “branch taken” prediction for the next branch.

To reverse-engineer the predictor, we use the HPCs to observe the mispredictions encountered by the BPU for each branch of a known sequence of branches. As the `perf_event_open` system call allows us to filter the hardware events by process, we are able to obtain a noise-free sequence of correct predictions and mispredictions, for each branch of a sequence. Note that the results of this phase depend on the micro-architecture we tested. For Broadwell, we are able to get traces of 0s and 1s (*i.e.*, 0 or 1 misprediction occurred). However, for older micro-architectures, we obtained traces largely consisting of 2s and 3s. To remove the noise and to match these traces to the ones obtained either for Broadwell or state machine models, we subtracted 2 to all measurements, thus obtaining 0s and 1s again.

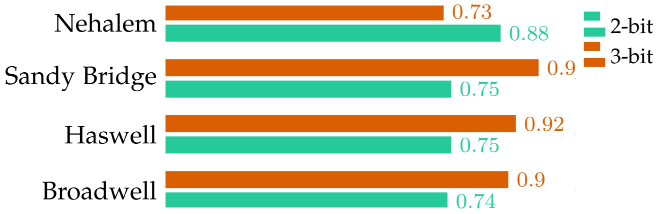


Fig. 1: Model accuracy on average for the 2-bit and 3-bit saturating counter state machines, for four micro-architectures.

In order to approximate the state machine that corresponds to the actual branch predictor, we first choose sequences of branches that have specific patterns (e.g., always taken, always not taken, and pattern repetitions). We start by forcing the predictor to be in a known initial state. We do so by executing branches with the same outcome, either taken or not taken, a large number of times. When the predictor is in a known state, we observe the number of branches with the opposite outcome to be executed to change the output of the predictor. For example, in the case of a 2-bit saturating counter, if the state machine is in the “strongly not taken” state, it takes two taken branches to reach a prediction of “branch taken”.

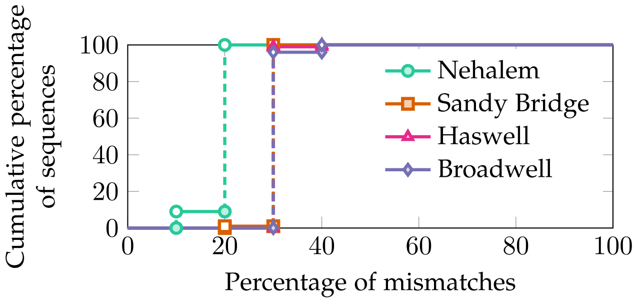
By scrutinizing several sequences of branches, we observe that a 2-bit saturating counter does not seem to model well the actual branch predictor for recent micro-architectures. Rather, we observe that a 3-bit saturating counter seems to be a better model.

3.2 Evaluation

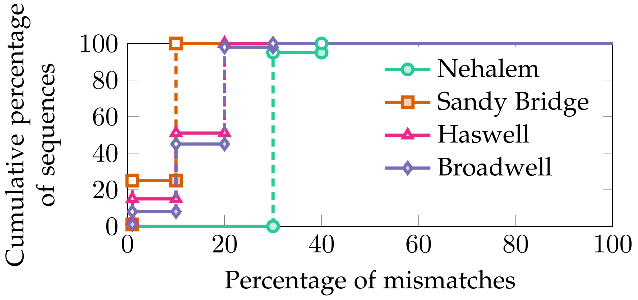
The evaluation of the reverse-engineering has been performed on the following setups: Intel Core i3-350M (Nehalem), Intel Core i3-2350M (Sandy Bridge), Intel Core i5-4210U (Haswell), Intel Core i5-5200U (Broadwell).

We now evaluate the accuracy of the different state machines (2-bit saturating counter and 3-bit saturating counter) to model the branch predictor of recent micro-architectures of Intel CPUs. We evaluate the accuracy by comparing sequences of mispredictions between one model and the actual mispredictions encountered by the branch predictor. We do not only seek to evaluate that the number of mispredictions is the same for each sequence, but rather that each misprediction encountered by the branch predictor is also encountered by the state machine. In other words, a perfect model gives the exact same sequence of mispredictions as the branch predictor, for any given sequence of branches. We evaluate the accuracy of each state machine with 10 000 sequences of 1 024 branches each. The sequence of branches is chosen randomly.

Figure 1 shows the accuracy of each state machine, for the four different micro-architectures in our setup. Two different trends appear. First, while a 2-bit saturating counter state machine is the best model for the Nehalem micro-architecture with 88% accuracy, this is not the case anymore from Sandy Bridge, where the accuracy of a 2-bit saturating counter drops to 75% accuracy, and a 3-bit saturating counter is a better model. Second, for the Sandy Bridge micro-architecture, we observe that a 3-bit saturating



(a) Predictor model: 2-bit saturating counter state machine.



(b) Predictor model: 3-bit saturating counter state machine.

Fig. 2: Empirical cumulative distribution functions, illustrating mismatches between mispredictions as predicted by one model, and actual mispredictions encountered by the branch predictor, for two predictor models and four micro-architectures. **The sooner a plot reaches 100%, the better the model.**

counter is a very close match, with 98% accuracy. The accuracy decreases a bit for the following generations (92% for Haswell, 90% for Broadwell), suggesting that the branch predictor has become a bit more complex.

These trends are also represented in Figures 2(a) and 2(b), where empirical cumulative distribution functions depict mismatches between mispredictions as predicted by one model, and actual mispredictions encountered by the branch predictor. With this representation, the sooner a plot reaches 100%, the better the model. For example from Figure 2, Sandy Bridge micro-architecture has less than 10% of mismatches between the predictor and the 3-bit saturating counter model for all sequences, whereas no sequence has less than 10% of mismatches between the predictor and the 2-bit saturating counter model, and only 1% of sequences have less than 20% of mismatches between the predictor and the 2-bit saturating counter. So the mismatch for a 2-bit predictor is more in case of Sandy Bridge compared to 3-bit predictor. In the remainder, we use 3-bit saturated counter as a model for the branch predictor in our side-channel attacks, since our attacks are performed on recent CPUs.

4 Attack Overview

In this section, we develop a general attack method that: (1) acquires HPC samples over the period of execution in an online phase, and (2) retrieves the secret using statistical techniques such as template building and template matching in an offline phase.

4.1 Threat Model

We target environments where hardware is typically shared between multiple users processes. This attack assumes that the adversary has the privilege to monitor HPCs, as HPCs have a restricted access since Linux 4.6. A legitimate user of a shared server system could be able to read the HPC misses of a similar user on the same server which she picks as her target. The hardware being shared, the mispredictions from one execution has an effect on the concurrent running process as well. In such a setting, the branch misprediction event counts can thus be observed over a target execution by HPCs. The attacker can achieve higher accuracy if he can pre-condition the branch predictor to a known state. However, as shown later, this is not a necessary requirement and attacks remains practical even without conditioning.

On the other hand, there are HPC enabled embedded devices which still allow access to HPCs from user level. Moreover, we emphasize the fact that the blinded implementations are not yet secure, unlike the general understanding of blinding. Alternative side-channel information like timing penalty for mispredictions could be used in place of HPCs. We believe that investigating information leakage due to HPCs is of crucial importance, as they provide very fine-grained information and fixing these attacks gives a better security margin.

The attack in its current form is not directly applicable across VMs because HPCs are obscured in this setup. Indeed, only a few software based performance events are allowed to be observed in a virtual environment setup using perf. However, mispredictions can be observed through other side channels such as timing [14]. The attack thus can be modified to a cross-VM scenario, with the same attack method but a different measurement procedure.

4.2 Practicality of our attack in SGX framework

Our attack is realistic in a SGX enclave running in hardware debug mode. As running an application in release mode on Intel SGX requires a commercial license agreement, this is out of scope of the paper. The implementation is such that there are two processes: (1) a victim process running in an enclave and (2) a spy process that observes the performance counter values through ioctl system calls from another enclave running in parallel to the victim enclave on the same processor core. The spy process measures the branch misprediction values across a dummy code snippet periodically using ioctl system calls in two different scenarios. First, the spy runs in the enclave along with all the background processes. The Gaussian distribution in Figure 3 in red shows the branch misprediction values as observed from HPCs for the dummy code snippet running in the spy enclave. Second, the Edward curve code (submitted version) gets executed as a victim process in another enclave which runs concurrently to the spy enclave (in blue in Figure 3). The HPC event counts for both the Gaussian distributions are observed from the spy enclave across the same dummy code snippet. This establishes the fact that the effect of branch misses can indeed be observed across enclaves, where a secret execution in an enclave has an effect on the performance counter values observed over a

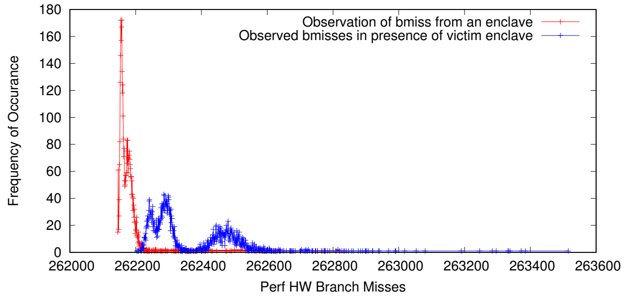


Fig. 3: Experiments from Intel SGX.

concurrent enclave. This makes our attack scenario realistic in the SGX framework.

4.3 Profiling Branch Mispredictions

To reverse-engineer the branch predictor, we performed synchronous measurements of branch misses over each iteration of the conditional block. However, such measurements may not be feasible for an attacker, as it would suppose to modify the executable run by the victim to insert measurements. Instead, we use the sampling mode of perf event counter values [15] to perform our attacks, without any modification of the victim program.

The perf object is instantiated with an event that is used as a sampler. For example, if the `instruction count` event is used as the sampling event, an interrupt is issued when the counter exceeds the value of the sample period parameter. We measure the branch miss event counter on each such interrupt. This provides perf handler to monitor with a fine granularity any cryptographic module running in the system sampled at a particular frequency of execution. The branch misprediction samples by the handler are *asynchronously* observed over the cryptographic module and there exists *no direct communication* between these two modules. This essentially means that the perf handler records the branch misprediction samples, but the samples being asynchronous in nature, it lack the trigger signals and make one-one correspondence more difficult with the concurrently running cryptographic module.

However, the counter values read by the interrupt handler for a very small sample period can be noisy, due to the overhead of the interrupt getting generated very frequently. The optimal sample period can be decided on the target machine, by testing with some dummy process. In our experiment platform (which we detail later), we found that the sampling becomes too noisy if we sample with a sampling period that is less than 50 instructions. This is typically because, if the sampling period is as low as say 100, after every 100 instructions, interrupts would be issued from the signal handler and read the count of another event, which introduces noise [15].

In our attack, we call the encryption process from the same script which instantiates the perf handler as in Figure 4a, however there is no direct communication between the two. A more optimistic scenario would be when the perf handler and the target cipher are run from two different scripts as we show in Figure 4b. As both the codes are running on the same processor core sharing the BPU, we observed that the distributions generated by the perf values indeed leak information. Figure 5 shows the result for the execution of such a scenario where the target Curve1174 scalar multiplication is repeatedly invoked and perf-statistics are noted using the perf handler. From Figure 5, it is evident that just by looking at the bunch of high sample points along the time scale one could distinguish the trace points that are getting affected by the execution of Curve1174.

In order to compare the traces obtained from two different scenarios we plot them such that they are aligned in the timeline with comparable sample periods. It is evident from Figure 6a that there is a similar pattern between the traces generated when the handler and the cipher are triggered by the same script versus two separate scripts. To highlight the similarity in the samples along the timeline, we compute the Pearson's correlation in windowed fashion (taking window of 5 trace points without loss of generality) and the windows which showed correlation higher than 0.8 (supposedly to be highly correlating) is plotted in Figure 6b. The sample traces as observed in Figure 6a have a high similarity if monitored intricately. Confirmatively, to measure the relative closeness of the two samples along time scale, the Bhattacharyya coefficient gives an approximate measurement of the overlap between two statistical samples. This coefficient is a popular choice in statistics theory for this particular problem. We measured that there is a correlation of around 0.78 (in the range $[0, 1]$) for the traces from two scenarios. Both Pearson and Bhattacharya coefficient metrics show that both traces are equally capable of launching a chosen attack based on the observed statistics. However, in the subsequent attack demonstration we refer to the scenario in Figure 4a, as here the noise level is lesser while the perf handler does not explicitly control the target cipher thus maintaining the practicality of the attack.

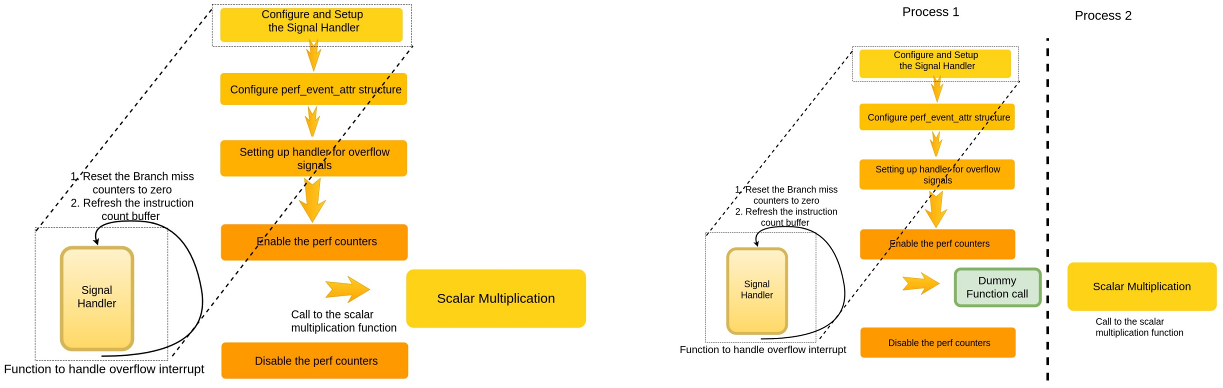
4.4 Template Attack on the Branch Predictor

We propose a strategy we call *Deduce* & *Remove* to target the scalar splitting and scalar randomization countermeasures, which randomize 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.

We assume a balanced ECC algorithm where the conditional execution is dependent on the n-bit scalar K. We consider m branch miss samples from the execution over K. Each branch miss sample is reported after a sample period of I instructions. Effectively, each sample of reported branch miss is affected by sample period I number of instructions thus, I is inversely proportional to m the number of reported branch miss samples.

In this paper, first we target the secret-dependent balanced conditional branching statements, which suffers from branch misprediction if the actual control sequence based on the secret scalar bit. In such cases, we choose I suitably such that $n/m = 2$. Moreover, considering a b-bit predictor, I is advised to be chosen such that $n/m \leq b$ ($b = 3$ in our case).

Second, we target the "double-and-add always" algorithm which does *not* have any conditional branching based


Configure and Setup the Signal Handler
Configure perf_event_attr structure
Setting up handler for overflow signals
Enable the perf counters
Scalar Multiplication
Call to the scalar multiplication function
Disable the perf counters
1. Reset the Branch miss counters to zero
2. Refresh the instruction count buffer
Signal Handler
Function to handle overflow interrupt
Process 1
Process 2
Dummy Function call

(a) Scenario using perf sampler in asynchronous sampling mode.

(b) Using perf sampler in asynchronous sampling mode from two different scripts.

Fig. 4: Perf handler execution scenarios.

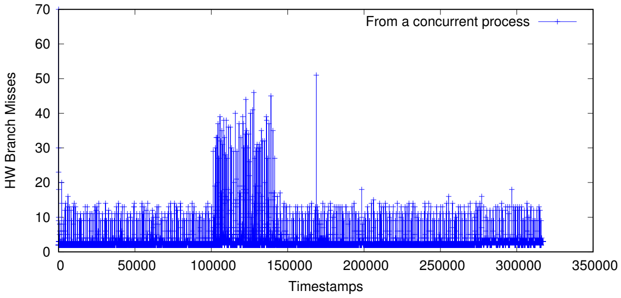


Fig. 5: Determining the perf samples corresponding to Curve1174 scalar multiplication in asynchronous sampling mode from a concurrent process

on the secret scalar. As shown later by supported experiments, such implementations often have data-dependent conditional modular reductions in the underlying field operations which can also be targeted for a successful attack. In such case, we choose I such that $m/n > 1$, and the template building and matching algorithms work efficiently for larger m/n values. In the following sections, we propose a generic methodology of template building and matching to collectively recover successive bits.

Offline Template Building for Each State of the Predictor

We propose the template building phase with the context of states of a generic b-bit predictor. A b-bit predictor has 2^b states, and each of them refers 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. A given sequence of taken and not-taken branches may exhibit a different number of mispredictions, depending on the state of the predictor at the start of the sequence. Thus the start state of the predictor could be crucial while modeling the simulated behavior of the branch mispredictions for construction of correct templates. As 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 the 2^t combination of bits are considered as templates and each of these traces contain $t \times m/n$ sample points. The template building phase is elaborated further in the following sections. At the end of the template building stage for each of start states of b- bit predictor, we obtain 2^b set of templates, each having 2^t templates of $t \times m/n$ sample points.

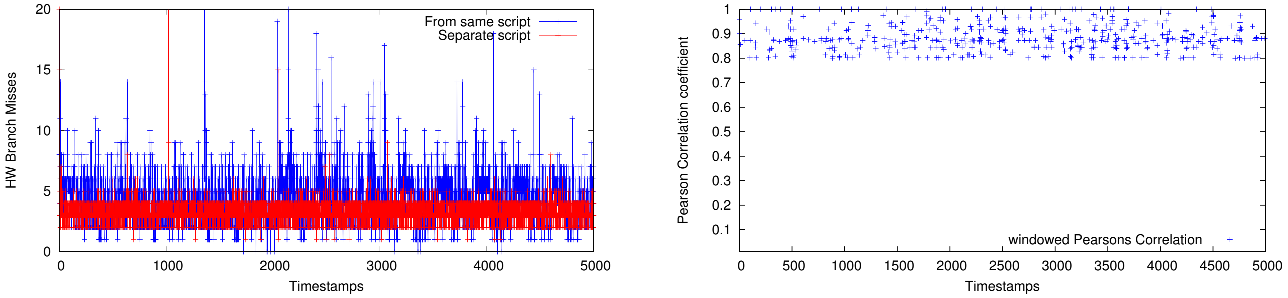
Offline Template Matching for an Unknown Trace

The template building phase is followed by a matching phase, where the sample trace collected for an unknown secret scalar is matched iteratively to the previously constructed templates. The matching phase is composed of the *Deduce* and *Remove* steps. In the *Deduce* step, we start matching from the Least Significant Bit (LSB) of the scalar multiplication. The matching can be done iteratively taking on a trace with s sample points ($s = t \times m/n$). 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 squares method. A set of templates having the least square 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 the same least square distance. Noise filtering is done in the *Remove* step. However, we noticed that the *Remove* step is device-specific and can also change with the algorithm. The template building and matching steps are further elaborated in Section 8.

At the end of the *Deduce* and *Remove* steps, the retrieved t bits decide 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 Blinding Countermeasures

In this section, we target three blinding countermeasures: scalar splitting, scalar randomization, and point blinding.



(a) Similarity in the perf samples corresponding to Curve1174 scalar multiplication in two different scenarios.

(b) Windowed Pearson correlation in two sample traces observed in two different scenarios

Fig. 6: Perf handler observations and their correlation in two different scenarios.

5.1 Attacking Scalar Splitting

The idea of scalar splitting as a DPA countermeasure for scalar multiplication appears in [5], [16]. Clavier and Joye [5] proposed a method to randomize the scalar K such that instead of computing KP, we compute $(K-r)P + rP$ where random r changes on every run. In this paper, we show that such secure implementation is still vulnerable to branch misprediction analysis. SPA resistant balanced scalar multiplications are computed on each of the splits separately. The SPA resistant scalar multiplication on split shares together result in DPA resistant algorithm.

The attack progresses from LSB to MSB. We now iteratively recover the bits of the secret scalar K, starting from LSB. We construct templates composed of trace pairs corresponding to all possible values of $(K-r)P$ and rP. The attack works in three major steps:

1) Acquire N pairs of traces corresponding to split scalar multiplications on $K-r$ and r 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.

Empirically rewriting the above discussion, for N pairs of traces $i \leq i \leq N$, a particular pair of trace TP_i is composed of a trace of $K-r$ which can be written as $T_{K-r} : t^0_{K-r}, \cdots, t^w_{K-r}, \cdots, t^M_{K-r}$, and the trace for r as $T_r : t^0_r, \cdots, t^w_r, \cdots, t^M_r$, where each of these window of traces contains s bits of trace point.

Combining the *Deduce* and the *Remove* step, the correct value of $K-r$ of the w^{th} window can be written as:

$$(K-r)_w : argmax_{(K-r)_w}\left[Pr[(K-r)_w | t^0_{K-r}, \cdots, t^w_{K-r}]\right]$$

$$(r)_w : argmax_{(r)_w}\left[Pr[(r)_w | t^0_r, \cdots, t^w_r]\right]$$

$$(K)_w = (K-r)_w + (r)_w$$

$$(K)_w = argmax_{(K)_w}[TP_1, \cdots TP_N].$$

5.2 Attacking Scalar Randomization

We now extend the attack to scalar randomization countermeasure. Although the generic attack principle stays the same, minor tweaks are needed when applied to this countermeasure. In scalar randomization, the algorithm computes $K' = K + r\#E$, where the secret scalar K is randomized every time with a new random number r and is multiplied with the order of the curve $\#E$.

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

1) Acquire N blinded scalar multiplication over $(K + r\#E)P$, for random values of r.
2) *Deduce*: For each of the N traces, we perform point-wise matching over s branch misprediction samples of t bits. It results in N candidates for t bits 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.

In simple words, we obtain templates of scalar multiplication over $k + r_1\#E$, $k + r_2\#E$ and $k + r_3\#E$ and perform template matching for them. Now we have 3 sets of candidate values say $S1$, $S2$ and $S3$ for three random split scalars. Now we take pair wise differences of each of these set elements, so it turns to $\{S1 - S2\}$, $\{S2 - S3\}$ and $\{S1 - S3\}$. Now if we again take pair-wise sum of the first two difference sets then it again becomes candidate values for $r1\#E - r3\#E$. If the template matching is successful with high probability, then these two sets would not be mutually exclusive.

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.

5.3 Attacking Point Blinding and Projective Randomization

The attack algorithm for scalar splitting and scalar randomization can be easily adapted against the point blinding and projective randomization countermeasure. Point blinding performs $K(P+R)$ instead of KP. Since the number of branch mispredictions does not directly depend on the point with which multiplication is done, the conditional branching statements if affected by the secret scalar K, then the branch misprediction traces can be used to retrieve the secret. Template matching returns 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.

The randomized projective coordinate countermeasure is similarly vulnerable if the control flow of execution is conditioned on the secret scalar bits. Both point blinding and projective coordinate randomization are effective countermeasure if the scalar bits are also randomized along with the input base point randomization.

6 Attacking Montgomery Ladder

6.1 Acquiring Traces

The following experiments are performed on an Intel Core i5-5200U CPU, running Ubuntu 16.04 with kernel 4.4.0-75-generic, and gcc version 5.4.0. We target the Montgomery Ladder implementation of ECC double-and-add scalar multiplication algorithm where the addition and doubling operations are conditioned on the blinded scalar bits. The sample points of the trace of branch misses obtained from sampling HPCs 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. Setting the `sampling_period` to 80 results in having one sample point affected by two bits of the scalar.

6.2 Building Branch Misprediction Templates

The success of our attack is highly dependent on how accurate are the built templates. 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, as this is the most preferred technique for conventional side-channel template attacks [17]. However, the noise affects the sample mean value, which therefore loses the correlation to the behavior of the 3-bit predictor. We choose to use the modal value, *i.e.*, the sample value which has occurred the most, as the candidate template point. Though we elaborate template matching for the full key in Section 8, in the following section we focus only on matching the LSB.

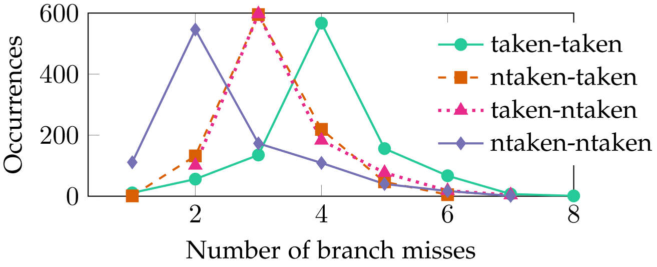


Fig. 7: Confusion in determining the LSB for scalar splitting.

6.3 Retrieving the LSB for Scalar Splitting

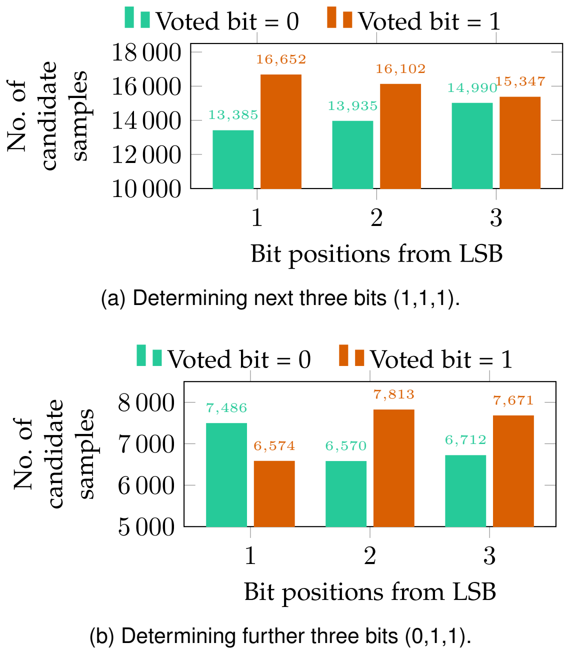
Among the observed sample points, the noisiest sample point is the first one, which is supposed to be affected only by the LSB, and the bit following the LSB, but often gets affected by the branch misses from instruction before the scalar multiplication starts to execute. The sampling being asynchronous to the underlying execution, this sampling noise has to be handled intelligently in order to diminish the chances of error.

Figure 7 shows a frequency analysis of the first branch miss samples observed over a set of random binary sequences of input. We observe that when the last two bits of the sequence both lead to *not-taken* branches, the distribution is shifted towards the left and exhibits overall lesser branch misses from the rest of the three cases. On the other hand, if the sequence is having two *taken* branches, it is suffering from the most branch misses and the distribution is shifted towards the right. The other two cases of combination of one *taken* and *not-taken* cannot be distinguished from each other simply by template matching.

To retrieve the LSB separately, we take the N pairs of samples for each of the split scalar and the random component such that we do not classify them if they are having sample values in the range $[3,4]$, as it can lead to an incorrect classification. For each of the N sample pairs:

- If sample point exhibits value <2, we classify both branches as *not-taken*, thus the bits as 11.
- If a sample point exhibits the value 2, we conclude that the branches are either both *not-taken* or *not-taken* followed by a *taken* branch, thus the bits as either 11 or 01.
- If a sample point exhibits the value 5, we conclude that the branches are either *taken* or *taken*, followed by a *not-taken* branch, thus the bits as either 00 or 10.
- If sample point exhibits a value >5, we classify both branches as *taken*, thus the bits as 00.

After finding the last two bits from all pairs out of the N, we take their pair-wise sum to conclude the LSB of the secret scalar K. We can also check whether each of the pairs leads to the same LSB of the secret scalar K.



(a) Determining next three bits (1,1,1).

(b) Determining further three bits (0,1,1).

Fig. 8: Determining 3 bits at a time for scalar splitting.

6.4 Iterative Template Matching for Scalar Splitting

We now iteratively retrieve the next scalar bits by applying the *Deduce* & *Remove* strategy once the LSB is known. In a first step, we perform template matching on the first $t = 10$ bits and deduce the candidate values which match respectively each of the N pairs of samples. Next, we take the pair-wise summation of such values, which 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 8a shows the majority voting of the next three bit positions from the LSB for the candidate values, where the correct bits (three consecutive 1s) clearly have a high majority. One striking observation is that this winning percentage gradually reduces for the next bits. The behavior of the branch misses has a huge similarity to 3-bit predictor characteristics, thus we decide only on the next 3 bits. Once we identify 3 bits, we retrace the split scalar values individually and remove the candidate values for both $K - R$ and R which do not tally the retrieved bits.

Knowing the LSB, we perform windowed template matching on the next t bits by sliding the window every time with the knowledge of 3 bits. Figure 8b reveals the next 3 bits as in $(0, 1, 1)$ based on adaptive template matching (*i.e.*, matching the templates after knowing the intermediate state of the predictor for the retrieved bits). We iteratively apply the same adaptive template matching to reveal the next bits without error.

6.5 Efficiency of *Deduce* & *Remove* Strategy on Scalar Randomization

Performing the check appearing in Section 5.2 can remove a significant amount of traces that are too noisy to leak any useful information. For scalar randomization, the online trace acquisition and template building phase work in the same way as for the scalar splitting algorithm. The attack algorithm in Section 5.2 is such that the secret scalar can be retrieved only when the full length of the blinded scalar has been retrieved. There is no iterative check to detect if an error occurred. In order to detect an error, we discussed a check mechanism that helps to correctly retrieve the first 3 bits of each random scalar and we update the predictor states after each 3 bits and proceed to retrieve the subsequent bits.

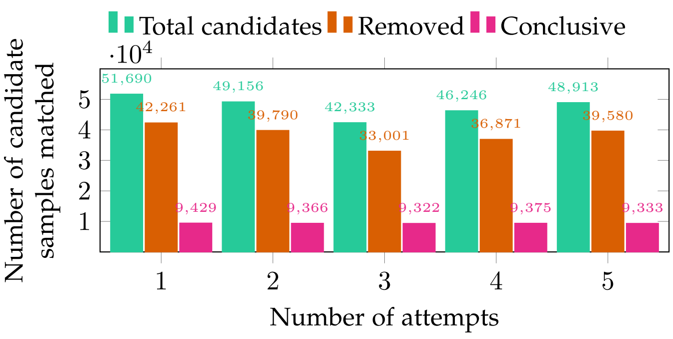


Fig. 9: Efficiency of *Deduce* & *Remove* strategy on scalar randomization.

The template matching phase in scalar randomization works on N misprediction traces, in our case we choose $N = 10\,000$. Each of these blinded scalars, after performing template matching by the least squares method, may have multiple candidate values to match the traces. The reason is simply that the acquired trace is noisy. Each blinded trace being random in nature, we denote the matched candidates for each trace as N sets $R_1, R_2, \cdots, R_N$, with $R_i = \{c_j : j \geq 1, c_j\text{'s are candidate matched templates of } t \text{ bits}\}$. We take three consecutive instances from the entire set. It is not infeasible to explore 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 5.2.

The efficiency of this *Deduce* & *Remove* strategy is illustrated in Figure 9. The number of correctly retrieved candidates is higher than 93%, *i.e.*, out of 10 000 separate random traces, more than 93% of the traces could identify the last 3 bits correctly among the 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 Attacking "double-and-add always"

In this section, we attack a more realistic and state-of-the-art implementation of Curve1174, an Edwards curve that has no obvious secret-dependent branching.[1]

7.1 Long Integer Multiplication Implementation

In our experiments, we consider the LSB-first double-and-add always algorithm [18]. The implementation of the algorithm is such that the execution is free from any straightforward secret scalar dependent branching. Unlike all existing branch misprediction attacks targeting the secret-dependent

[1] The code supporting this attack is publicly available: https://github.com/SBIIT/Branch-Attack-on-Curve-1174.

branching of public key algorithms, in this paper we illustrate a data-dependent branching key retrieval attack.

The intuition behind the attack is that the executing operation remains the same for both bit values 0 or 1, but the operands differ depending on the secret scalar bit. More specifically, if $b = 0$, then $R_1 = 2R_1$, $R_1 = R_1 + R_0$ is executed, else, $R_0 = 2R_0$, $R_0 = R_1 + R_0$ is executed. Curve1174 follows a *unified formula* where both of these doubling and addition operations are performed using the same set of equations. Each of these point additions takes two points (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2) as input and computes the third point as (X_3, Y_3, Z_3). For point doublings, the same point is fed twice as input. These point additions are implemented using a series of Long Integer Multiplication (LIM). The field multiplications which form the components of the addition and doubling operation are implemented in an arithmetic co-processor with a LIM followed by a reduction. The detailed algorithm for performing a LIM can be found in Algorithm 1 in [19]. Each of the LIM has a modular reduction step which is typically carried out using a data-dependent conditional structure. The modular reduction step is thus carried out in each of these multiplication operations and conditional reduction statements are dependent on the input data as well as the secret scalar. The secret scalar bit can assume either one of the two values, the input operands to the LIM thus vary based on the secret scalar, which inherently affects the branch misprediction profiles of modular reductions.

7.2 Attack Methodology

In previous attack algorithm we targeted the conditional branching on the secret, thus considered multiple scalar bit retrieval at a time. This algorithm does not include any direct control flow moderation based on the secret bits, thus we go one step further to understand the template building and matching of intermediate data based conditional structure of modular reduction of the double and add unified formula for Curve1174.

Let us assume that the adversary seeks to identify the i^{th} bit of the blinded scalar. In the process, the adversary acquires N independent blinded traces of branch misses. The assumption for this attack is that the previous $(i-1)$ blinded bits have been retrieved for each of the N traces. The adversary guesses each i^{th} bit of the j^{th} blinded scalar sequence ($j \in N$) having known all previous $(i-1)$ bits.

For a guess of either 0 or 1, the adversary simulates the branch misses corresponding to all the subsequent modular reduction function calls for the i^{th} bit of the j^{th} blinded scalar. To do so, the adversary uses the b-bit predictor model and generates a trace of simulated mispredicted branch sequences, denoted $bm\_sim^0_{i,j}$ for a guess of 0, and $bm\_sim^1_{i,j}$ for a guess of 1. The branch misprediction samples observed from the perf ioctl calls corresponding to the i^{th} bit and j^{th} blinded scalar are denoted as $bm\_perf_{i,j}$. The relative ordering of the branch misses as observed from the perf call $bm\_perf_{i,j}$ shows a correlation to the relative ordering of the branch misses for the correct guess.

In noisy environments, this relative correlation is not very reliable to craft a highly successful attack. Thus instead we demonstrate a more realistic approach using the described logic of simulated branch mispredictions in the following subsection.

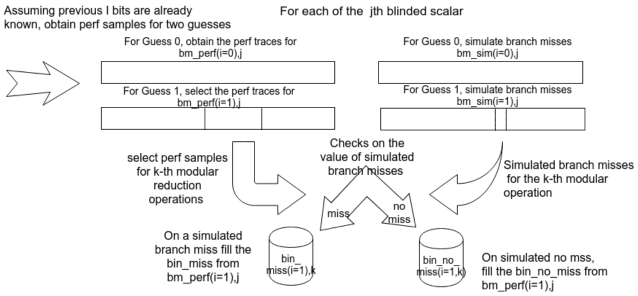

Assuming previous I bits are already known, obtain perf samples for two guesses
For each of the jth blinded scalar
For Guess 0, obtain the perf traces for bm_perf(i=0),j
For Guess 1, select the perf traces for bm_perf(i=1),j
For Guess 0, simulate branch misses bm_sim(i=0),j
For Guess 1, simulate branch misses bm_sim(i=1),j
select perf samples for k-th modular reduction operations
Checks on the value of simulated branch misses
Simulated branch misses for the k-th modular operation
miss
no miss
On a simulated branch miss fill the bin_miss from bm_perf(i=1),j
bin_ miss(i=1),k
bin_no_ miss(i=1,k)
On simulated no mss, fill the bin_no_miss from bm_perf(i=1),j

Fig. 10: Template Building Phase.

Acquiring Traces

In the previous attack, we targeted the secret-dependent conditional branching, thus there were less affected sample points. In this experiment we target all the data-dependent operations involved in long integer multiplications that are performed while each point addition operation executes. Each long integer multiplication approximately executes with 12 000 instructions.

The measurement procedure in our experiments does not have direct communication between the targeted curve module and the handler code. As a result of this, there is no specific trigger signal available to the adversary to understand the correspondence between a misprediction and the secret bit. Without loss of generality, we made an initial estimation of the number of instructions required to execute each point addition operation by taking the expectation of repeatedly running it. Typically, we observed 600 misprediction samples (with a sampling period of 1 600 instructions) for a set of two point addition operations for a particular secret scalar bit.

The trace collection phase is the most time consuming phase of the attack. In our experiments, this took at the most less than an hour. The later part of the analysis of template building and template matching is purely computational and iterative in nature.

7.3 Template Building on the Acquired Traces

In the template building phase, we consider 600 sample points which are responsible for a particular secret scalar bit. We apply a windowing technique to determine the correspondence of the observed samples and the simulated branch mispredictions. Building template for data-dependent branches in Curve1174 requires two sets of branch misprediction traces: (1) the simulated mispredicted traces from 3-bit predictor for each modular reduction operations involved in a point addition operation, (2) the perf samples corresponding to the same set of inputs.

The adversary targets each of the j blinded scalars in an iterative manner. The i^{th} bit of blinded scalar is guessed and both 0 and 1 is appended to the already known $(i-1)$ bits, separately for each of the j blinded scalar sequences. We define $bm\_perf_{(i=0),j}$, $bm\_perf_{(i=1),j}$ as branch misprediction samples affected by the guessed bit, and the operation is performed over all j sequences and known $(i-1)$ bits for two separate guesses. This essentially generates $2 \cdot j$ sequences and we separately take the ioctl branching samples. Similarly, we have the simulated branch misses from the 3-bit predictor $bm\_sim^0_{i,j}$ and $bm\_sim^1_{i,j}$ for both guesses as discussed earlier. We illustrate the template building phase in Figure 10. The steps to build a template are as follows:

1) For each k instances of the modular reduction operation, we apply a windowing technique to $bm\_perf_{(i=0),j}$ and $bm\_perf_{(i=1),j}$ to identify approximately how many samples are responsible for each modular reduction operation.
2) Now for $guess = 0$, we consider $bm\_perf_{(i=0),j}$ and $bm\_sim^0_{i,j}$, and separately build template points based on whether or not they suffer from a branch miss.
3) For $guess = 0$, we separately consider templates from the samples in $bm\_perf_{(i=0),j}$ for each of the k modular reduction operations involved for point addition for each of the j sequences. We construct two bins for each of the k modular reduction operations based on whether they have a simulated branch miss at particular modular reduction step $bm\_sim^0_{i,j}$. Now we fill the $bin\_miss_{(i=0),k}$ with samples from $bm\_perf_{(i=0),j}$ for the k^{th} particular modular reduction, if there has been a simulated branch miss in $bm\_sim^0_{i,j}$. Otherwise, we fill $bin\_no\_miss_{(i=0),k}$ if there is no misprediction.
4) We separately construct templates taking the mode of the distributions of each of these constructed bins as described in the previous discussions of template building.

At the end of this step, we have $2 \cdot k$ separate bins for all the k modular reduction operations considering all the j sequences where all the i^{th} bit has been guessed to zero. A similar construction can be performed with i^{th} bit being guessed as 1.

7.4 Template Matching to Unknown Traces

In this phase, the adversary observes the acquired traces over the unknown i^{th} bit for each of the blinded scalar sequences. For each of these sequences, a template matching is performed to the templates built in the previous phase.

We assume that the adversary knows the previous $(i-1)$ bits of the blinded secret scalar. He can also guess the unknown bit and simulate the branch misprediction to identify branch mispredictions for the modular reduction operations. At this stage, the adversary takes branch misprediction samples for each of the k modular reduction operation, and matches with the templates built for both guesses using the least squares method (LSQ).

The simulated branching for guessed bit 0 may cause a misprediction. If there has been a simulated misprediction for $guess = 0$, then the unknown samples are matched with the templates corresponding to $bin\_miss_{(i=0),k}$, else, the unknown samples are matched with the templates corresponding to $bin\_no\_miss_{(i=0),k}$. At this step, the least squares method returns a distance for k^{th} modular reduction for templates built with $guess = 0$ from either the mispredicted template or the template with no misprediction. Similarly, the template matching is applied on each k operations for $guess = 1$. We then sum up all the respective distances obtained by template matching for all k modular reduction operations for $guess = 0$ and $guess = 1$ respectively. Finally, we choose the i^{th} unknown scalar bit to be the one having least sum of distances from the LSQ.

TABLE 1: Retrieval of few target bit positions across a secret scalar with scalar splitting countermeasure with 100 traces.

Bit position	Time (s)	Scalar bit	Retrieved
12	142.73	1	yes
45	206.42	0	yes
60	241.08	0	yes
89	302.17	0	yes

7.5 Attacking Scalar Splitting and Randomization

Template building and matching phases as discussed in the previous subsections are highly efficient to retrieve the unknown blinded scalar bits for all blinding countermeasures. These two phases constitute the *Deduce* step of our attack. Unlike the secret-dependent branching results, the sample points obtained in data-dependent branching attack provide us various modular operations to check the efficiency of our attack. This multiple template matching step to deduce a single bit inherently makes this attack much efficient than the previous one. The conditions for the *Remove* step is exactly same as has been applied previously.

The conditioning of branch prediction was used in the previous attack as described in Section 6.3, though in the context of the current attack the conditioning of the perf handler is impractical. This is because for the current attack scenario the branches are executed as a result of the iterated data dependent operations. Thus mimicking a particular branch sequence as a result of pure data dependent instruction needs crafting of data in such a fashion to enforce occurrence of such branches. Thus we suggest that the conditioning is not mandatory to demonstrate our attack, though it should be noted, if a particular adversary is capable of conditioning the branch prediction hardware to one of his desired state then it makes the side channel less noisy and easier to analyze.

The success rates of the attack on scalar splitting are illustrated in Figure 11. The figure shows the percentage of sequences that has successfully revealed the blinded scalar for 4 consecutive bit positions over 10 000 sample traces. Individual template matching on each of the split scalars reveals the split scalar and the random mask efficiently. Thus, the *Deduce* step seldom gives a wrong probability of matching and the overall success probability improves. In our experiments we revealed 4 bits consecutively on Curve1174 long integer multiplication over split scalars and applied the final check using the *Remove* step to prune out the wrongly deduced scalars.

Similarly, this attack is highly efficient to retrieve blinded scalar bits in a scalar randomization countermeasure as illustrated in Figure 12. In this figure, we plot the success, failure and inconclusive percentages of the 10 000 blinded scalar getting retrieved. Since the template matching algorithm reveals one bit at a time, the *Remove* step for scalar randomization cannot be applied to each bit. This is because the *Remove* algorithm for scalar randomization would fail to remove any bits wrongly getting predicted by the *Deduce*

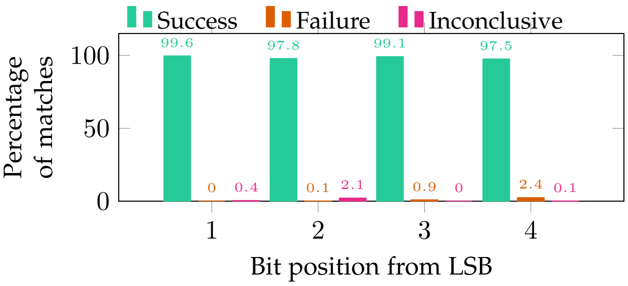


Fig. 11: Probabilities of bit retrieval for scalar splitting in Curve1174 over 10000 traces.

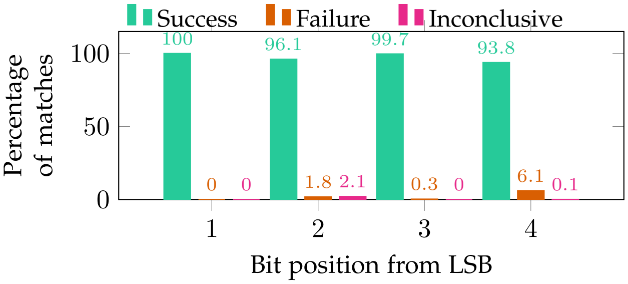


Fig. 12: Probabilities of bit retrieval for scalar randomization in Curve1174 over 10000 traces.

step. In this case, the template matching step is performed on t multiple bits subsequently and the *Remove* algorithm is applied on these multiple retrieved bits together (identically to the previous attack described in Section 6.5).

We present the experimental results on random target bits of the secret scalar in Table 1 for scalar splitting countermeasure. The attack succeeds in recovering any of the 128 secret scalar bits correctly, assuming that all the previous bits are correctly retrieved in the previous iterations of the attack. The table also shows the time required for the attack analysis, which includes the time to form the templates for each of the split scalar and the random traces and matching the template with the observed traces. Since the traces for higher bit positions are longer, the time taken for the analysis increases with bit position.

The attack methodology described here is much stronger than the attack demonstrated on secret-dependent branching for Montgomery Ladder implementation and this is because the number of samples getting affected for data-dependent branching is observed over many modular reductions and thus larger in number than the secret-dependent branching.

8 Investigating Template Matching

In the discussion provided in the previous section, modal values for the corresponding trace points were elected as the reference template value for the particular classification. The primary reason behind this selection was that the noise component in the individual trace points were not uniformly affected. Some of the trace points were highly affected by noise and for those trace points, branch misprediction values were majorly varying from the signal component. On the other hand, the rest of the trace points were having only the signal component. If the distribution of these individual trace points were considered to be Gaussian, the mean value of these classified templates were showing almost no difference since the noise component in the noisy set of traces was hiding the signal component of the less noisier traces. Remarkably, on the other hand there are clear differences if the highest occurring value(ie, the modal) is considered as the template point, since in this case the signal component remains unaffected from the effect of noisy samples. Based on this observation from the individual branch misprediction tracepoints, we have considered modal values in place of mean values to be the representative of their respective template class. Thus following this template building phase, we supported the results with a Least Squared template matching phase which worked with significantly high accuracy.

The original form of template based analysis on power traces as presented in [17], [20], [21], works with a construction of mean trace vector and a noise co-variance matrix. The noise co-variance matrix holds a major significance on the success of the template matching since it captures the correlation of the noise components of the consecutive trace points. The noise covariance matrix grows quadratically with the increase of trace points. We have reworked this template creation and matching steps using the same equations as in [17], [20]. The probability of template matching for a test trace t, with a template class defined with a mean trace m and noise covariance matrix (C) on assumption of data d_i and key k_j is calculated as:

$$p(t;(m,\mathcal{C})_{d_i,k_j}) = \frac{exp(-\frac{1}{2}\cdot(t-m)'\cdot C^{-1}\cdot(t-m))}{\sqrt{(2\cdot\pi)^T\cdot det(C)}}$$

The accuracy of template matchings for the individual templates using the above equation are shown in Figure 13 in green bars. The accuracy of individual template being correctly matched using the noise covariance matrix calculation are higher than other techniques.

The matching step as described in [20] involves computation of an inverse of the co-variance matrix which could be difficult to compute if there are more points in the trace matrix. Thus [20] introduced the concept of reduced templates which performs template matching with the following equation.

$$p(t;m) = \frac{1}{\sqrt{(2\cdot\pi)^{N_{IP}}}}\cdot exp\Big(-\frac{1}{2}\cdot(t-m)'\cdot(t-m)\Big)$$

Constructing reduced templates from the branch misprediction samples and the accuracy of template matching using reduced templates is illustrated in Figure 13 in orange bars. The reduced template matching accuracy is quite good compared to the best accuracy observed via mean and noise covariance template matching in green.

Next, we replaced the mean vector of the reduced template to the highest observed tracepoint value, *i.e.*, the median of the tracepoint values and repeated the exact experiment. This time the template matching step exactly matches with the modal value selection and matching using LSQ step that we were performing earlier in the paper. This particular mode reduced template matching in magenta performs comparably to the mean reduced templates in

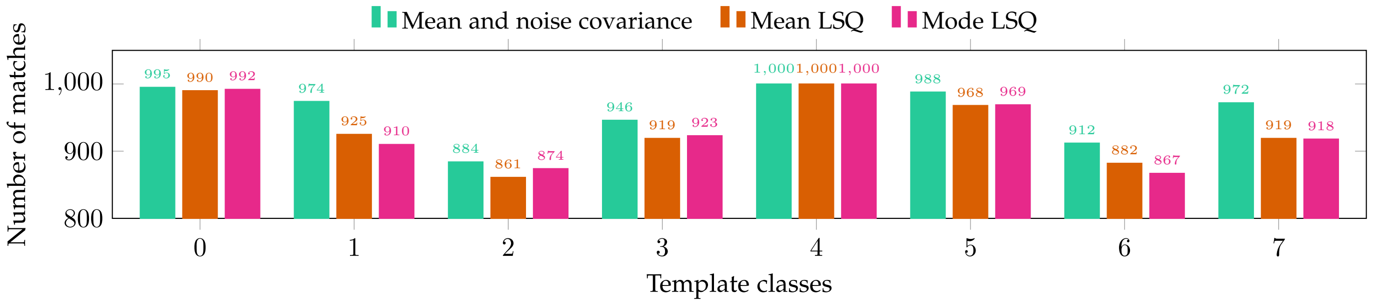


Fig. 13: Number of successful template matching for scalar splitting in Curve1174.

orange and sometimes better than the mean reduced LSQ. Figure 13 thus illustrates the comparison across different template matching techniques and all the three techniques work with significant accuracy. Thus we can conclude from these results that Mean LSQ and Mode LSQ are equally compatible as an estimate for successful template matching.

9 Attacking other Cryptographic implementations

9.1 Template Attack on Curve25519

In this section, we extend our attack description to the scalar multiplication computation on the Montgomery Curve25519. An efficient way for performing scalar multiplication on Curve25519 is to use homogeneous projective coordinates. A more efficient implementation further uses the x-coordinate-only representation of points as described in [22]. We target a MSB-first Montgomery powering ladder algorithm realizing the scalar multiplication implementation. The target implementation involves a sequence of point swappings based on the secret bit followed by point additions and point doublings. We have adapted the Montgomery ladder algorithm for Curve25519 using the set of addition doubling formula on projective coordinates as shown in Algorithm 1 in the paper [23]. The iterative algorithm as in [23], computes same set of equation on every iteration. First, based on the secret scalar bit a conditional swap of the input is performed, then it is followed by addition and doubling equations. The implementation has no secret dependent branching for performing addition and doubling steps and the conditional swapping is realized using constant time swap as in Algorithm 7 in [24].

Though there are no secret dependent conditional branching in the implementation, the branchings corresponding to the modular reduction operation with respect to the prime $2^{255} - 19$ are non-constant time. The number of reductions and the correction step are dependent on the reduction based on the new prime $2^{255} - 19$ and the implementation is similar to Curve1174 discussed in Section 7. The template building phase results are presented in Figure 15, and the success to different template matching techniques are illustrated in Figure 14. The template matching accuracy for Curve25119 as appears in Figure 14 is significantly high. Since we target a MSB first algorithm, the *Remove* step would not work on the partial bits. So in this case, the *Remove* step can only be applied after all the bits of blinded scalar values are matched separately. This is similar to the scalar randomization method described in Section 7.5.

9.2 Attacking RELIC Implementations

There are a few implementations of Curve1174, an Edward curve, available as a part of public cryptographic libraries. In this section, we illustrate the applicability of our branch prediction based template attack to the publicly available implementations of Curve1174 and Curve25519 in RELIC [6]. RELIC is a modern cryptographic library which provides several ECC and pairing implementations, implemented with emphasis on efficiency and flexibility [7]. There are a number of algorithmic choices which a developer can use while executing the particular curve operations. To illustrate that the library is vulnerable to branch misprediction traces, we perform a simple experiment where we incorporate a template-based analysis targeting the branch mispredictions for Montgomery Ladder implementation. Figure 16a represents branch misprediction samples from HPCs for the Montgomery Ladder algorithm with curve parameters of Curve25519 for two different guesses of the bit value.

The RELIC library comes with a number of test codes along with the implementation. We target one such test code for Curve1174. The template building and matching steps are similar to the attack demonstrated in Section 7.3. Like the previous attack methodology, we incorporate an iterative attack on each bit of the blinded scalar. The algorithm under analysis being a MSB first algorithm, the Figure 16b illustrates the success rate of the scalar blinded bits retrieved correctly using the template matching technique.

10 Countermeasures

At the *application level*, the first and the most obvious countermeasure suggested by the previous works in literature is to implement the scalar multiplications such that the control flow of the execution is independent of the secret scalar K. However, in this paper we showed that the data-dependent branches can still be attacked by the adversary and thus are inefficient to thwart branch misprediction attacks. Thus we suggest that the code needs to be constant-time such that it is free from both secret and data-dependent branching. Projective randomization and point blinding would be vulnerable if we target the naïve Montgomery ladder with branches depending on secret. However, when secret dependent branches are removed, the present attack does

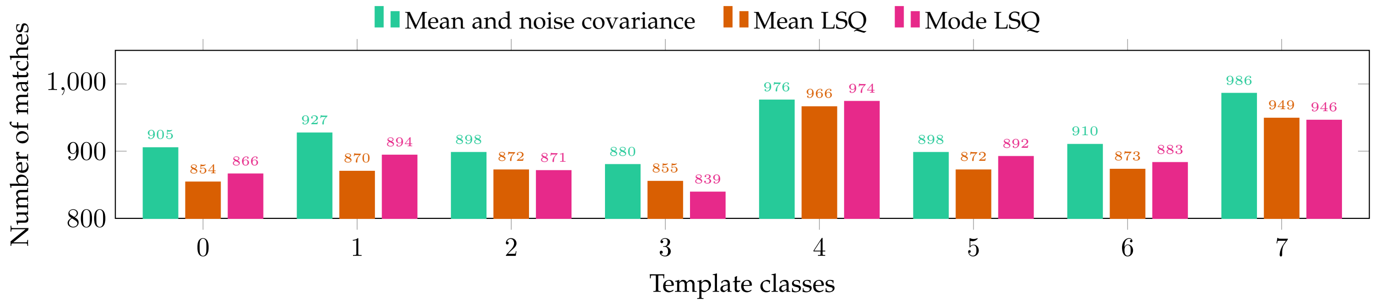


Fig. 14: Number of successful template matching for scalar splitting in Curve25519.

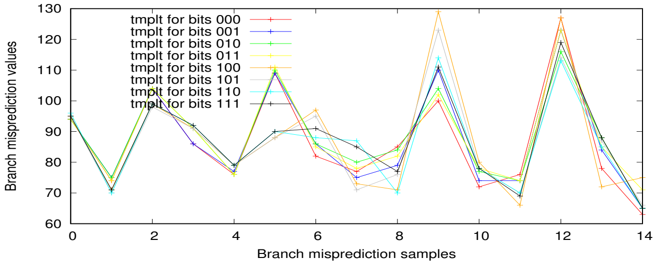
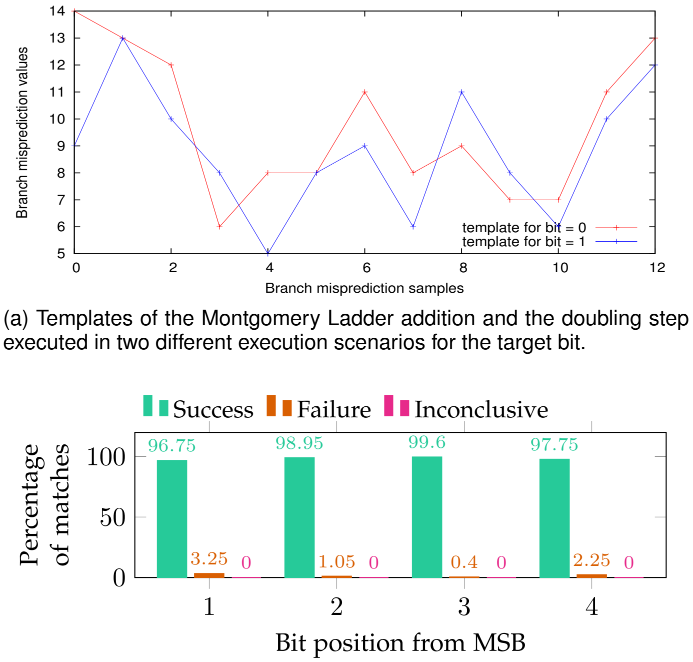


Fig. 15: Difference in templates of the scalar multiplication on Curve25519 for three subsequent bits from the LSB.



(a) Templates of the Montgomery Ladder addition and the doubling step executed in two different execution scenarios for the target bit.

(b) Probabilities of bit retrieval for scalar randomization in RELIC implemented Curve1174.

Fig. 16: Determining 3 bits at a time for scalar splitting.

not directly apply. Thus the attack presented in Section 6 still find relevance in presence of such point randomization since the control flow of execution depends directly on secret bits. While the attack in Section 7 performs double-and-add always, this countermeasure is still applicable in such case. However the fact that the modulo operations have branches depending on the data, can be exploited using cross-correlating branch miss traces. For doubling operation there are many pairs with same long integer multiplication inputs (and thus have the same branch pattern).

Another countermeasure is to randomize the state of the predictor intermediate to the execution. This can be achieved, e.g., at the *system level*, by introducing random branching executions in parallel to the execution of the victim library. The objective is to randomize the state of the predictor to inhibit the *Deduce* step of the attack.

Randomization in several layers of the algorithm and measurements can only make the attack more difficult, as the adversary may be able to obtain more fine-grained traces. This brings us again to the open challenge of rethinking the structure of the branch predictor at the *hardware level*, such that they are inherently secure against these attacks.

11 Related Work

11.1 Microarchitectural Side-Channel Attacks

Acıiçmez et al. [25] first observed that the penalty for mispredicted branches in number of clock cycles is a side channel able to identify the data-dependent operations of the public-key cryptographic system. A further improved version of this attack [26], [27] has also been carried out with proper knowledge of the underlying hierarchical Branch Target Buffer (BTB) architecture of the target system. The work of Acıiçmez et al. [25] has been extended by Bhattacharya and Mukhopadhyay [3], using the HPCs present in recent processors. Lee et al. [28] used the Last Branch Record feature of Intel CPUs that stores information on recent branches to attack SGX enclaves. Molnar et al. [29] proposed techniques for implementing binary exponentiation algorithms without requiring branch instructions. However, studying the use of HPCs to exploit cipher codes implemented with branch statements is vital, as there still exist several standard implementations using branches. Moreover, some implementations without conditional branches have been subjected to side-channel attacks other than timing [30], [31]. Alternately, other side-channels were also used to exploit secret dependent leakage in ECC [32].

In addition to targeting cryptographic algorithms, side channels on the BPU can be used to perform other types of side-channel attacks, such as deriving kernel and user-level ASLR offset [33], or covert channels [34], [35]. Recently, a side-channel attack on directional branch predictors was proposed [14]. The recent Spectre [36] attack also exploits branch misprediction and requires a fine-grained knowledge of the branch prediction unit.

Finally, HPCs have already been utilized for attacks. Uhsadel et al. [37] exploited L1 and L2 D-cache miss counters to attack the AES T-Table implementation found in OpenSSL.

11.2 Reverse-Engineering CPU Components

Few papers have tackled the task of reverse engineering the BPU. In particular, Milenkovic et al. [38] 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. Uzelac and Milenkovic [39] reverse-engineered more details of these structures, including interactions between different structures, focusing on the Pentium M architecture. Prior work mostly focused on the *structures* of the branch predictor rather on the predictor itself, using microbenchmarks. As these works were published respectively on 2004 and 2009, they also target older processors.

Apart from the BPU, other micro-architectural components have been reverse-engineered. Maurice et al. [40] reverse-engineered the last-level cache addressing scheme of modern Intel CPUs, using information from the uncore HPCs. Pessl et al. [41] reverse-engineered the DRAM mapping functions on both x86 and ARM platforms, using a timing side channel.

12 Discussion and Future Work

We have targeted single trace attacks by successfully applying template attack on the partial scalar components and trying to retrieve them iteratively with the branch misprediction traces. However, Howgrave-Graham and Smart [42] have shown that even with a lower number of known bits from multiple random scalar components could be subjected to Lattice based attacks to recover the entire key. The paper presents a detailed discussion on how to use lattice attacks to break ECDSA when a small numbers of bits if many split random keys are known. This scenario directly applies to our paper as well. The full key recovery has been adapted from the seminal work by Coppersmith [43], which uses the LLL algorithm to solve the univariate and bivariate modular polynomial equations. This opens a new direction to our present paper.

Moreover, the open source implementation of isogeny-based cryptography [44] is inherently composed of conditional statements which could be exploited by observing branch mispredictions. This is an interesting research direction that we would like to explore in the future.

13 Conclusion

Information leakage from the branch predictor is known to pose a serious threat to asymmetric key cryptographic algorithms containing a conditional statement depending on the secret. In this paper, we presented the first template attack on the branch predictor. We initially performed the reverse engineering of the branch predictor of various micro-architectures, from Nehalem to Broadwell, and showed that recent generations of branch predictors can be modeled as a 3-bit saturating counter. Subsequently, we used this model to attack the DPA secure implementations of ECC, which were believed to be secure against branch prediction side-channel attacks. We also tackled Curve1174, Curve25519, where the double-and-add always algorithm implementation is free from conditional branching on the secret scalar.

Acknowledgments

This work was supported in part by the TCS Ph.D Fellowship in IIT Kharagpur and Temasek Laboratories in NTU Singapore, and in part by the project ANR-19-CE39-0007 MIAOUS of the French National Rsearch Agency (ANR).

References

[1] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware,” *Journal of Cryptographic Engineering*, vol. -, pp. 1–27, Dec. 2016.

[2] P. C. Kocher, “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems,” in *CRYPTO ’96*, 1996, pp. 104–113.

[3] S. Bhattacharya and D. Mukhopadhyay, “Who watches the watchmen?: Utilizing performance monitors for compromising keys of RSA on intel platforms,” in *CHES*, 2015, pp. 248–266.

[4] J. Coron, “Resistance against differential power analysis for elliptic curve cryptosystems,” in *CHES*, 1999, pp. 292–302.

[5] C. Clavier and M. Joye, “Universal exponentiation algorithm,” in *CHES*, 2001, pp. 300–308.

[6] D. F. Aranha and C. P. L. Gouvêa, “RELIC is an Efficient LIbrary for Cryptography,” https://github.com/relic-toolkit/relic.

[7] N. E. Mrabet and M. Joye, *Guide to Pairing-Based Cryptography*. Chapman & Hall/CRC, 2016.

[8] “OpenSSL,” https://github.com/openssl/openssl.

[9] “libgcrypt,” https://github.com/gpg/libgcrypt.

[10] “BouncyCastle,” https://github.com/bcgit/bc-java/tree/master/jce/src/main\\/java/javax/crypto.

[11] “mbedTLS,” https://github.com/ARMmbed/mbedtls.

[12] O. Aciiçmez, Ç. K. Koç, and J. Seifert, “On the power of simple branch prediction analysis,” *IACR Cryptology ePrint Archive*, vol. 2006, p. 351, 2006. [Online]. Available: http://eprint.iacr.org/2006/351

[13] O. Aciiçmez, J. Seifert, and Ç. K. Koç, “Predicting secret keys via branch prediction,” *IACR Cryptology ePrint Archive*, vol. 2006, p. 288, 2006. [Online]. Available: http://eprint.iacr.org/2006/288

[14] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, “BranchScope: A New Side-Channel Attack on Directional Branch Predictor,” in *ASPLOS’18*, 2018.

[15] Ubuntu Manuals, “perf_event_open-set up performance monitoring,” http://manpages.ubuntu.com/manpages/wily/man2/perf_event_open.2.html, 2017.

[16] J. Danger, S. Guilley, P. Hoogvorst, C. Murdica, and D. 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*, 2016, pp. 374–386.

[17] S. Chari, J. R. Rao, and P. Rohatgi, “Template Attacks,” in *CHES*, 2002.

[18] M. Joye, “Highly regular right-to-left algorithms for scalar multiplication,” in *CHES*, 2007, pp. 135–147.

[19] A. Bauer, E. Jaulmes, E. Prouff, J.-R. Reinhard, and J. Wild, “Horizontal collision correlation attack on elliptic curves,” *Cryptography and Communications*, vol. 7, no. 1, pp. 91–119, 2015.

[20] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer, 2007.

[21] O. Choudary and M. G. Kuhn, “Efficient template attacks,” in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2013, pp. 253–270.

[22] P. L. Montgomery, “Speeding the pollard and elliptic curve methods of factorization,” *Mathematics of computation*, vol. 48, no. 177, pp. 243–264, 1987.

[23] P. Koppermann, F. De Santis, J. Heyszl, and G. Sigl, “X25519 hardware implementation for low-latency applications,” in *2016 Euromicro Conference on Digital System Design (DSD)*. IEEE, 2016, pp. 99–106.

[24] C. Costello and B. Smith, "Montgomery curves and their arithmetic," *Journal of Cryptographic Engineering*, vol. 8, no. 3, pp. 227–240, 2018.

[25] O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert, "Predicting Secret Keys Via Branch Prediction," in *CT-RSA*, 2007, pp. 225–242.

[26] O. Aciiçmez, S. Gueron, and J.-P. Seifert, "New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures," in *IMA Int. Conf.*, ser. Lecture Notes in Computer Science, S. D. Galbraith, Ed., vol. 4887. Springer, 2007, pp. 185–203.

[27] O. Aciiçmez, J.-P. Seifert, and Çetin Kaya Koç, "Micro-Architectural Cryptanalysis," *IEEE Security & Privacy*, vol. 5, no. 4, pp. 62–64, 2007.

[28] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *26th USENIX Security Symposium*, 2017.

[29] D. Molnar, M. Piotrowski, D. Schultz, and D. 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, Revised Selected Papers*, 2005, pp. 156–168.

[30] P. Fouque and F. Valette, "The doubling attack - why upwards is better than downwards," in *CHES*, 2003, pp. 269–280.

[31] A. Chakraborty, S. Bhattacharya, T. H. Dixit, C. Rebeiro, and D. Mukhopadhyay, "Template attack on SPA and FA resistant implementation of montgomery ladder," *IET Information Security*, vol. 10, no. 5, pp. 245–251, 2016.

[32] E. Nascimento, Ł. Chmielewski, D. Oswald, and P. Schwabe, "Attacking embedded ecc implementations through cmov side channels," in *International Conference on Selected Areas in Cryptography*. Springer, 2016, pp. 99–119.

[33] D. Evtyushkin, D. V. Ponomarev, and N. B. Abu-Ghazaleh, "Jump over ASLR: attacking branch predictors to bypass ASLR," in *MICRO 2016*, 2016.

[34] ——, "Covert channels through branch predictors: a feasibility study," in *Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2015*, 2015.

[35] D. Evtyushkin, D. Ponomarev, and N. B. Abu-Ghazaleh, "Understanding and mitigating covert channels through branch predictors," *TACO*, vol. 13, no. 1, pp. 10:1–10:23, 2016.

[36] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *S&P*, 2019.

[37] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2008, pp. 59–67.

[38] M. Milenkovic, A. Milenkovic, and J. Kulick, "Microbenchmarks for determining branch predictor organization," *Software: Practice and Experience*, vol. 34, no. 5, pp. 465–487, 2004.

[39] V. Uzelac and A. 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)*, 2009, pp. 207–217.

[40] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *RAID*, 2015.

[41] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: exploiting DRAM addressing for cross-cpu attacks," in *USENIX Security Symposium*, 2016.

[42] N. A. Howgrave-Graham and N. P. Smart, "Lattice attacks on digital signature schemes," *Designs, Codes and Cryptography*, vol. 23, no. 3, pp. 283–290, 2001.

[43] D. Coppersmith, "Finding a small root of a univariate modular equation," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1996, pp. 155–165.

[44] Microsoft, "Sidh v3.2," https://github.com/microsoft/PQCrypto-SIDH.

Sarani Bhattacharya is currently joining as a post-doctorate researcher at KU Leuven Belgium. She has just finished her Ph.D from IIT Kharagpur, India. Her current research interests include micro-architectural attacks and countermeasures, secure system design and computer architecture security.

Clémentine Maurice is a researcher working for CNRS at IRISA (Rennes, France). Among other topics, she is interested in micro-architectural covert and side channels in commodity computers and servers, and reverse-engineering processor parts.

Shivam Bhasin is currently a Senior Research Scientist and Principal Investigator at PACE Lab, Nanyang Technical University, Singapore since 2015. His research interests include embedded security, trusted computing and secure designs.

Debdeep Mukhopadhyay received his PhD from Dept. of Computer Science and Engineering, IIT Kharagpur in 2007, where he is presently an Associate Professor. His research interests include cryptography, VLSI of cryptographic algorithms, hardware security and side channel analysis.