Online Detection and Reactive Countermeasure for leakage from BPU using TVLA

Bhattacharya, Sarani; Bhasin, Shivam; Mukhopadhyay, Debdeep

2017

Bhattacharya, S., Bhasin, S., & Mukhopadhyay, D. (2018). Online Detection and Reactive Countermeasure for leakage from BPU using TVLA. 2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), 155-160.

<https://hdl.handle.net/10356/88917>

<https://doi.org/10.1109/VLSID.2018.54>

© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The published version is available at: [<http://dx.doi.org/10.1109/VLSID.2018.54>].

Downloaded on 26 Apr 2023 20:15:21 SGT

Online Detection and Reactive Countermeasure for leakage from BPU using TVLA

Sarani Bhattacharya
Department of Computer Science and Engg
IIT Kharagpur
sarani.bhattacharya@cse.iitkgp.ernet.in

Shivam Bhasin
Temasek Laboratories
NTU Singapore
sbhasin@ntu.edu.sg

Debdeep Mukhopadhyay
Department of Computer Science and Engg
IIT Kharagpur
debdeep@cse.iitkgp.ernet.in

Abstract—Branch Prediction Units (BPUs) of computing systems have been targeted by several side channel analysis of public key encryptions. In recent years, performance counters have been used as a side channel source for the branch mispredictions which can be used to attack ciphers with user privileges. In this paper we propose an online leakage detection tool Branch-Monitor for branch misprediction traces which does an online detection of the leakage and raises an alarm if there exists a significant difference in the distribution of branch misses for selected inputs. The Monitor triggers a randomization module on detecting such leakage which effectively runs a software module to confuse the branch predictor unit such that it inherently prevents the information leakage. We have practically validated our detection module on Intel systems and is easily scalable to other platforms and processors.

I. 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.

Hardware Performance Counters are a set of special-purpose registers storing the counts of hardware-related activities within the microprocessor. These counters are affected by the internal activities of the processor and hence can be utilized as a source of information leakage. In [1], the authors exploited these HPCs as side-channels for time based cache attacks. HPC L1 and L2 D-cache miss counters have been exploited as side-channels in [1] for performing cache attacks on symmetric-key algorithms, like AES as in [2].

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 this paper, we propose Branch-Monitor which is an online

detection tool for attacks and threats which exploit the information leakage from the Branch Predictor Unit (BPU).

The first attack targeting the BPU appears in [3], where the penalty for mispredicted branches in number of clock cycles is observed as side-channel to identify the data dependent operations of the public-key cryptosystem. 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. A further improved version of this attack [4], [5] has also been carried out with proper knowledge of underlying hierarchical Branch Target Buffer (BTB) architecture of the target system. In [6], the authors introduced a new covert channel to perform secret communication between the Trojan and the spy processes which exploits the residual state of the dynamic branch predictor behavior of the system. While in [7], authors describe an implementation of a Contention-based Covert channel. An attack has been developed in [8] to derive kernel and user-level ASLR(Address Space Layout Randomization) offset which exploits the BTB collisions between the branch instructions. In [9], [10], it was first established that branch misses from HPCs can reveal the secret key in RSA. In [11], techniques for implementing binary exponentiation algorithms without requiring branch instructions have been proposed. However, the study of using Hardware Performance Counters (HPCs) to exploit the cipher codes implemented with branch statements is vital because there still exist several standard implementations using branches (as in OpenSSL [12], [13]).

Our approach needs no knowledge of the implementation of the algorithm under consideration. We perform hypothesis testing as proposed in the Test Vector Leakage Analysis(TVLA) [14] methodology on the output misprediction traces to check for data dependent leakage. In [15] TVLA is introduced as a reliable, quick and easy test to detect leakage between two specific distributions. This paper clearly shows the formulation of TVLA and validates their analysis on various modes of AES. A better analysis and leakage detection on any higher orders was formalized in [16]. The authors in [17] improved the original formulation of TVLA to make it more robust to environmental noise and observed that paired- t test works significantly well under such circumstances. While in a more recent paper [18], the authors have a fast leakage assessment methodology which can detect leakage orders of

magnitude faster than the original TVLA formulation. This approach computes histograms for separate classes and separate time samples and then performs the statistical moments calculation on these histograms. In all of these works, the leakage analysis has been applied over power measurement traces, but in [19] this statistical t -test is applied to the timing side channel. The implementation design described in the paper determines whether a particular software module run in constant time or not.

In this paper, we propose an online branch misprediction leakage detection tool as Branch-monitor which can detect leakages by observing the branch misprediction traces and starts a defence mechanism if the distributions are significantly different. The contributions of our paper are:

- In this paper, we present a thorough analysis of the branch misprediction traces, leakage from those traces and detecting these leakage points using the Branch-monitor implementation.
- We evaluate our analysis on two setups of Intel processors and show the effectiveness of Branch-Monitor to detect leakage using a significantly lower number of traces.
- Following this, we propose a randomization module which is triggered as and when the Branch Monitor detects the leakage and, the experiments show that the countermeasure thus proposed to confuse the BPU hardware is efficient to prevent the leakage.
- Lastly, we analyze the timing overhead of the normal process computation in the presence of the randomization module.

II. PRELIMINARIES

In this section, we provide a background on some key-concepts, which include some implementation algorithms for public-key ciphers and some well-known branch predictors which have been subjected to attack.

A. 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 $(k_0, k_1, \dots, k_i, \dots, k_{n-1})$. 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 $(b_0, b_1, \dots, b_{n-1})$.

- 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, i.e.,

addition gets performed. Since this is a normal flow of execution, the branch is considered as not-taken i.e., $b_j = 0$ 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, the branch is taken i.e., $b_j = 1$.

Thus for any `if-else` block, we consider the respective branch statement to be `not-taken` if the `if` conditional statement 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.

B. Test Vector Leakage analysis (TVLA)

Test Vector Leakage Analysis(TVLA) was first proposed in [14] to identify if a distribution of power trace from a cipher implementation is statistically different from the another set of traces using the Welch's t -test in which the test statistic follows a Student's t distribution. The t statistic calculation takes power traces in two groups and performing the calculation independently on both the groups. Both the groups take power traces in two subsets A and B.

- N_A and N_B can be assumed as the size of the subsets A and B respectively.
- Compute X_A the average of all the traces in group A, X_B the average of all traces in group B.
- S_A be the standard deviation of the traces in group A and S_B is the sample standard deviation of the traces in group B.
- Computes the t -statistics for the particular trace as:

$$\frac{X_A - X_B}{\sqrt{\frac{S_A^2}{N_A} + \frac{S_B^2}{N_B}}} \quad (1)$$

Each trace is composed of an array of measurements sampled across different timestamps. The average and sample standard deviations of the traces are calculated for each sample point vertically across the set of traces. Thus after calculating mean and standard deviation they are also vectors over the same points in time.

The same analysis is repeated for the other group for their subsets. If at any point in time the t -test statistic exceeds $+/- 4.5$ for both of the groups, the device fails the security test. This effectively claims that the device under test leaks information such that the two separate subsets are significantly distinct from each other and the null hypothesis is rejected.

In the next section, we understand the leakage from the branch prediction unit and in the following sections we propose the new online leakage detector to thwart information leakage through branch misprediction HPC event counts.

III. LEAKAGE FROM BRANCH PREDICTION UNIT

The values of the event counters which are available to user level processes through Linux perf utility actually leak a significant amount of information about the concurrently running processes in the system. The event counters get affected by the concurrent processes running in the system, and surprisingly they have a huge impact on the event counts as well. We emphasize on this with a simple experimental scenario, where two users share the same hardware and have two different processes running.

- Process 1: An unprivileged user running multiplication operation.
- Process 2: A privileged user performing exponentiation which is dominated mostly with conditional if-else statements.
- The unprivileged user observes Perf statistics for the user level multiplication process concurrent to the privileged exponentiation process.

As illustrated in Figure 1, it has been observed on an Ubuntu 16.04 system that there is an unexpected increase in the number of branch misses in the perf stat of the user process while the execution of exponentiation is performed. If the unprivileged user runs the multiplication process and observes the number of branch misses for the multiplication process then it encounters around 200 – 220 branch misses. There has been a sharp increase in branch misses on and from 30 ms in Figure 1 which is only getting affected due to the exponentiation process being run by the privileged process. As in Figure 1, we observe that the time from which there is an increase in the number of branch misses (as observed by the user process), coincides with the time when the exponentiation algorithm begins. This experiment shows:

- An unprivileged user process residing on the same system as the privileged process can gain access to sensitive information of the privileged, or more generally any other user's process execution.
- Thus the increase or decrease of branch misses of a privileged process can be monitored by some spy process having only user level privileges.

IV. BRANCH-MONITOR: IMPLEMENTATION AND ANALYSIS

We propose Branch Monitor as a watchdog to the information leakage from the hardware performance counters. In this paper, we demonstrate that the information leakage for a shared user platform from the branch misprediction hardware is significant and our proposed detection tool can detect the leakage by applying a statistical test on the captured traces of the branch mispredictions using the perf 'ioctl' calls. We tabulate the steps of the detection as follows:

Fig. 1. Abrupt increase in branch miss observed by unprivileged user due to exponentiations from privileged process

- Branch-Monitor tool observes the branch misprediction traces for an executable and performs an online t-statistic test over time.
- For each set of points in the trace, the statistic is modified accordingly, and an alarm is raised if the value exceeds the threshold $+or - 4.5$.

In [9], the authors showed that the information leakage from the branch misprediction traces is much powerful to leak the secret key bits in as low as 100 traces. Our Branch-monitor detection tool can detect data-dependant leakage in only 40 traces.

In the next subsection, we explain the trace collection procedure for branch misprediction.

A. Obtaining branch misprediction traces using perf ioctl calls

Branch-Monitor is an online branch-misprediction analysis tool which observes the branch mispredictions from Hardware Performance counters (HPCs). The measurement procedure with Branch-Monitor uses a dummy code and observes the misprediction traces over the dummy code for the concurrently running encryption or decryption operation. Granularly observing the branch misses over each iteration of the if-else block of the code under consideration is only possible when the monitor is very closely observing the underlying algorithm, or the tool has the control of the underlying code. Since our tool Branch-Monitor is assumed to perform an online analysis, we present a much practical solution based on sampling perf event counter values using ioctl calls. The idea is such that the perf object is instantiated with an event which is used as a sampler. For example, if instruction count event is used as the sampling event, branch miss event is measured from HPCs for the particular sampling period.

B. Implementing the Branch-Monitor

We have implemented our prototype for Branch-Monitor as illustrated in Figure 2 using a c-code which runs on all variety of systems. We have targeted the Edward-1174 curve implementation written with long integer multiplications in c. The code computes scalar multiplication using the secret scalar by performing doubling for each bit and conditional addition for the bits which are set. For the statistical t-test to detect leakage, we need two sets of inputs to be fed to the algorithm under test. The test setup is described as follows:

- Branch-Monitor collects branch misprediction event counts over a periodic interval of instruction count which is entirely handled by the perf ioctl system calls.
- The measurements of branch misses are observed on a dummy code snippet, while the secret scalar multiplication is getting performed concurrently in background.
- We obtained two sets of misprediction traces where the concurrently running Edward curve scalar multiplication was once provided a fixed basepoint as input and in the second case the input basepoint is varied randomly.
- Online mean and variance of the two sets are calculated separately and the t statistic value is measured.

Figure 3(a) and 3(b) represents two scenarios where the leakage is detected in as low as 30 traces on an Intel-i5-5200U processor running Ubuntu 16.04 LTS, and a particular part of the trace is observed to leak more as the number of traces are increased. The t statistic is calculated over each point on the trace, and the figures clearly show that both of the traces leaks information since the t statistic is having a value above 4.5 and below -4.5. The x-axis in the figures represent the individual sampling point of a branch misprediction trace and the y-axis values represent their appropriate t -statistic value. We have replicated our experiments on a comparatively older processor Intel-i5-3470 running Ubuntu-12.04 LTS, and the leakage detection scenario is illustrated in the Figures 4(a),(b). The figures clearly show that there are multiple points of leakage where the t statistic crossed the range(4.5,-4.5).

In the following section, we propose an interesting countermeasure to prevent this form of leakage.

V. BPU RANDOMIZATION BY BRANCH-MONITOR TOOL

We present a practical countermeasure to thwart these forms of leakages through branch misprediction information by randomizing the internal states of the branch prediction hardware. The BPU being shared by all processes of the same processor core, the effect of one process is predominant on the concurrently running execution on the same processor core. This property has been used by the attackers in [9] to attack secure implementation. In this paper, we use the same philosophy to prevent the leakage from these branch misprediction event counters. Branch-Monitor, as shown in Figure 2 being an

Fig. 2. Branch-Monitor design proposal

online detection tool, the BPU randomization countermeasure is not triggered unless an alarm is raised.

- The monitor raises an alarm when the online detection mechanism encounters the branch misprediction leakage.
- On an alarm, the Branch-Monitor starts its defence mechanism of randomizing the state of the BPU such that the branch misprediction traces observed with such randomized design loses its correlation with the key.

The randomization module design is very easy to implement and highly effective to randomize the internal state of the BPU such that the information monitored over the BPU cannot be used to reconstruct the states of the secret computation. The Branch-monitor on detecting a leakage triggers on a particular randomization module within the code, which is typically written using a conditional if-else structure of code. The conditional if-else structure on execution results in conditional branching instructions, which are made to be dependent over randomly generated 0/1 binary sequences. Since these random sequences bear no correlation with the secret scalar and the randomization module executes concurrent to the cipher module execution, the internal BPU states are modified by both the cipher module and the randomization module. This prevents the leakage through the event counters, which normally leaks a huge amount of information through these tools which are available in the user space.

The experiment as performed on Ubuntu 16.04 LTS is illustrated in Figure 5(a),(b) which plots the leakage in the presence of the BPU randomization. The leakage detection is performed using HPC event counts with the randomization being performed along with the cipher execution. Figure 5(a) shows that the randomization works really well such that the t -statistic is well within the range ± 4.5 for 2000 traces. We just made the test run overnight and figure 5(b) plots the leakage from 20000 traces, which also shows that the t statistic value is not crossing the region of -4.5 to 4.5 . Again, we replicate the same experiment for the Intel-i5-3470 running Ubuntu-12.04 LTS and the Figures 6(a),(b) shows that for a significantly higher number of traces the leakage is not exceeding the safe limit thus preventing the leakage.

A. Impact of this randomization on timing side-channel

Fixing a side-channel leakage sometimes leads to a leakage through some other channels. In our design, the Branch-Monitor incorporates randomization to the Branch Predictor intermediate state by running a piece of code concurrent to a sensitive cipher execution. This randomization being due to concurrent executions, result in a number of context switches of the processes and thus can affect the execution time of the sensitive application. We plotted the frequency distribution of the execution time of the cipher implementation in two different scenario: when only the cipher implementation is running and second case when both the Branch Monitor and the randomization modules are running. The distributions as shown in Figure 7 absolutely overlap on each other and thus there is indeed minimal or no overhead in execution time on the running when Branch-monitor is allowed to run on

(a) TVLA on 30 traces

(b) TVLA on 200 traces

Fig. 3. Test Vector Leakage Analysis on Edward-1174 implementation on Ubuntu 16.04 in Intel Core i5-5200

(a) TVLA on 100 traces

(b) TVLA on 2000 traces

Fig. 4. Test Vector Leakage Analysis on Edward-1174 implementation on Ubuntu 12.04 in Intel Core i5-3470

(a) TVLA on 2000 traces in presence of randomization

(b) TVLA on 20000 traces in presence of randomization

Fig. 5. Test Vector Leakage Analysis on Edward-1174 implementation on Ubuntu 16.04 Intel Core-i5 5200U with concurrently running randomization module

(a) TVLA on 2000 traces in presence of randomization

(b) TVLA on 20000 traces in presence of randomization

Fig. 6. Test Vector Leakage Analysis on Edward-1174 implementation on Ubuntu 12.04 Intel Core-i5 3470 with concurrently running randomization module

Fig. 7. Histograms of execution time for Edward-1174 curve while detection (without randomization) and with randomization shows minimal overhead on execution time of these two scenarios

Fig. 8. Variation of timing-TVLA with increasing number of traces

the system. Next, we extended our experiments to test that the countermeasure proposed here leaks information through timing channel or not. We performed the t -statistic test on the execution time of the Edward-1174 curve implementation when the randomization module is also getting executed in parallel to the cipher implementation. Figure 8 illustrates how the t -statistic is changing as the number of traces are increased, and even after observing for 25000 traces the TVLA value indicates that there is no leakage through timing channels. This makes our claim even stronger, that the BPU randomization is an effective countermeasure in preventing the leakage and the timing-TVLA result adds up to its confidence.

VI. CONCLUSION

In this paper, we present an effective detection mechanism for the cipher implementations with respect to branch misprediction leakage. The branch misprediction traces are less noisy and much powerful to known conventional side channels and thus requires fewer traces to reveal secret. Thus we needed a detection mechanism which could detect the leakage much faster and start the defense mechanism so as to prevent further leaking of sensitive information. We validated all our experiments for the proposed, very efficient Branch-Monitor on Intel platforms. In this paper, we have only addressed the leakage assessment for branch mispredictions, but there exists a group of performance counter events which are equally vulnerable. Extending this analysis for the leakage assessment test of the whole bunch of performance counters would be the future objective of this research.

REFERENCES

- [1] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *FDTC*, L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J.-P. Seifert, Eds. IEEE Computer Society, 2008, pp. 59–67.
- [2] J. Bonneau and I. Mironov, "Cache-Collision Timing Attacks Against AES," in *CHES*, ser. Lecture Notes in Computer Science, L. Goubin and M. Matsui, Eds., vol. 4249. Springer, 2006, pp. 201–215.
- [3] O. Aciicmez, Çetin Kaya Koç, and J.-P. Seifert, "Predicting Secret Keys Via Branch Prediction," in *CT-RSA*, ser. Lecture Notes in Computer Science, M. Abe, Ed., vol. 4377. Springer, 2007, pp. 225–242.
- [4] O. Aciicmez, 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.
- [5] O. Aciicmez, J.-P. Seifert, and Çetin Kaya Koç, "Micro-Architectural Cryptanalysis," *IEEE Security & Privacy*, vol. 5, no. 4, pp. 62–64, 2007.
- [6] D. Evtushkin, D. V. Ponomarev, and N. 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*, 2015, pp. 5:1–5:8.
- [7] D. Evtushkin, 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.
- [8] D. Evtushkin, D. V. Ponomarev, and N. 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*, 2016, pp. 1–13.
- [9] S. Bhattacharya and D. Mukhopadhyay, "Who watches the watchmen?: Utilizing performance monitors for compromising keys of RSA on intel platforms," in *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, 2015, pp. 248–266.
- [10] —, "Formal fault analysis of branch predictors: attacking countermeasures of asymmetric key ciphers," *Journal of Cryptographic Engineering*, vol. 2016, pp. 1–12.
- [11] 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, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, 2005, pp. 156–168.
- [12] Y. Yarom and N. Benger, "Recovering openssl ECDSA nonces using the FLUSH+RELOAD cache side-channel attack," *IACR Cryptology ePrint Archive*, vol. 2014, p. 140, 2014.
- [13] B. B. Brumley and R. 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*, 2009, pp. 667–684.
- [14] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, "A testing methodology for sidechannel resistance validation," *Non-invasive Attack Testing Workshop*, vol. 2011.
- [15] G. Becker, J. Cooper, E. DeMulder, G. Goodwill, J. Jaffe, G. Kenworthy, T. Kouzminov, A. Leiserson, M. Marson, P. Rohatgi, and S. Saab, "Test vector leakage assessment (tvla) methodology in practice," in *International Cryptographic Module Conference 1001,2013*, 2013.
- [16] T. Schneider and A. Moradi, "Leakage assessment methodology - A clear roadmap for side-channel evaluations," in *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, 2015, pp. 495–513.
- [17] A. A. Ding, C. Chen, and T. Eisenbarth, "Simpler, faster, and more robust t-test based leakage detection," in *Constructive Side-Channel Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers*, 2016, pp. 163–183.
- [18] O. Reparaz, B. Gierlichs, and I. Verbauwhede, "Fast leakage assessment," *IACR Cryptology ePrint Archive*, vol. 2017, p. 624, 2017.
- [19] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, 2017, pp. 1697–1702.