Exploring Micro-architectural Side-Channel Leakages through Statistical Testing

Sarani Bhattacharya and Ingrid Verbauwhede
imec-COSIC, ESAT, KU Leuven
{sarani.bhattacharya, ingrid.verbauwhede}@ esat.kuleuven.be

***Abstract*—Micro-architectural side-channel leakage received a lot of attention due to their high impact on software security on complex out-of-order processors. These are extremely specialised threat models and can be only realised in practise with high precision measurement code, triggering micro-architectural behavior that leaks information. In this paper, we present a tool to support the inexperienced user to verify his code for side-channel leakage. We combine two very useful tools- statistical testing and hardware performance monitors to bridge this gap between the understanding of the general purpose users and the most precise speculative execution attacks. We first show that these event counters are more powerful than observing timing variabilities on an executable. We extend `Dudect`, where the raw hardware events are collected over the target executable, and leakage detection tests are incorporated on the statistics of observed events following the principles of non-specific *t*-tests. Finally, we show the applicability of our tool on the most popular speculative micro-architectural and data-sampling attack models.**

I. Introduction

This paper addresses one of the most pertinent questions in the area of micro-architectural security. Computer architecture gets refined over the years with the performance optimization as its sole objective with little effort towards thorough understanding of the security implications of these architectural optimizations. The primary aim of this work is to investigate the systematic mitigation against software visible micro-architectural side-channels. Our primary objective is to identify micro-architectural features that can be exploited in practice and hence should be included in security models.

Motivated by the increasing need to integrate leakage resilient micro-architectural design solutions into processor designs, in this paper we are seeking a standard approach that enables a fast, reliable and robust evaluation of the micro-architectural vulnerability of the target executable, being oblivious of the actual implementation and the underlying hardware. As we are already aware, attack-based testing is not straightforward. It requires intricate designing of the attack model, requires precise knowledge of the target micro-architecture and moreover, does not cover all possible attack vectors. Thus in this paper, *the goal is to establish a robust testing methodology to assess the micro-architectural vulnerability of a target executable.*

Thus we follow by the concepts of statistical testing which has been adapted in invasive hardware side-channel leakage detection over the years [1], [2]. Statistical leakage testing has been shown as a powerful tool for detecting leakages and uncovering vulnerabilities in side-channel attacks. The authors in [3] have proposed an open source tool `Dudect` which also adopts statistical testing to detect timing variability in cryptographic software executables.

The speculative execution leakages thrive on the footprint of the out-of order execution on the shared architectural resources. These architectural resources such as cache memory, branch predictor, Line Fill Buffers, are the architectural components which are affected because of the out of order execution. Thus, here we need the architecture specific monitors. We propose to extend `Dudect` which not only uses timing, but also uses Hardware Performance Counters for monitoring the architectural events.

Contributions of the paper

- First, we show that using timing variability is not sufficient towards leakage detection of micro-architectural exploits.
- Next, we incorporate the more precise measurements from the raw hardware performance events into `Dudect`. These hardware event monitors provide more precise information over execution footprints of the executable on the underlying hardware compared to timing measurements.
- We show the effectiveness of leakage detection tests on cryptographic libraries and on the most popular speculative execution exploits.

II. Statistical testing in a nutshell

Statistical leakage detection techniques work on the fundamental observation of whether two sets of data are significantly different from each other. The test indicates leakage, if the two distributions are statistically distinguishable from each other. This is realised in practise by performing the Welch's *t*-test[1], where test statistic follows a Student's *t* distribution [1].

In order to perform this test, we require two set of sample observations, say two probability distributions $\mathcal{Q}_0$ and $\mathcal{Q}_1$: Sample $\mathcal{Q}_0$: μ_0, s_0^2, n_0 and Sample $\mathcal{Q}_1$: μ_1, s_1^2, n_1.
In the *t*-test, we determine a probability to examine the validity of the `Null Hypothesis` denoted as $\mathcal{H}_0 : \mu_0 = \mu_1$. For the sake of simplicity, usually a threshold on the *t*-statistic ($\mid t \mid > 4.5$) is defined to reject the `Null Hypothesis`. The *t*-test statistic is calculated as, $t = \frac{(\mu_0 - \mu_1)}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}}$,

[1]https://en.wikipedia.org/wiki/Welch%27s_t-test

978-3-9819263-5-4/DATE21/©2021 EDAA

where μ_0, μ_1 and s_0^2, s_1^2 and n_0, n_1 represent the sample mean, sample variance and cardinality of each set.

This forms a robust and reliable technique to distinguish two distributions and found its importance in leakage detection tests in context of hardware side-channel attacks [1].

A. Why are statistical leakage detection tools important?

Statistical Leakage detection techniques consider the behavior of the process under consideration as a black-box. The black-box structure of the leakage detection procedure is highly useful for regular users, because this does not demand any knowledge of implementation details of the target process. The leakage tests are also platform independent and thus make them portable. In addition, the test suite is constructed in a way that it does not require any specific details of the underlying hardware of the target platform.

III. Evaluating Dudect with timing observations for micro-architectural leakages

A. Dudect and understanding Timing variability

Dudect [3] in its original form, is a very compact tool that can be used to test timing variability in cryptographic functions in an efficient way. Dudect does not rely on static analysis but on statistical distinguishability of execution timing measurements.

Though, it should be noted, that execution time is a manifestation of a gamut of events occurring simultaneously in a micro-processor. An application can execute with different execution times on two different platforms depending on a variety of factors. The factors include compiler optimization, latency in accessing various components in the computer architecture, difference in timing due to computational latency, memory optimizations, interrupts and many more. Thus apparently, running statistical leakage detection tests using timing values reveal that there is a significant difference in behavior in timing, which could be because of one of the several contributing factors. Thus timing leakages successfully indicate the distinguishability in behavior of two classes of input, but unfortunately does not help to pinpoint the source.

B. Micro-architectural Leakage Models

As discussed earlier, timing leakages from the micro-architecture can be a function of several parameters: Branch prediction, Hyperthreading, Memory Technology, Cache Architecture. In this section, we will look into such examples of the timing leakage detection on micro-architectural events.

Experimental Setup:

The experiments in this paper have been conducted on *Intel Core i7-3770 CPU* and *Intel Core i5-8265U CPU*, both *running Ubuntu 18.04.5 LTS* and *Intel Core i5-5200U running Ubuntu 16.04 LTS*. The timing measurements have been observed using the ReaD TimeStamp Counters (RDTSC) calls as supported in the Dudect tool.

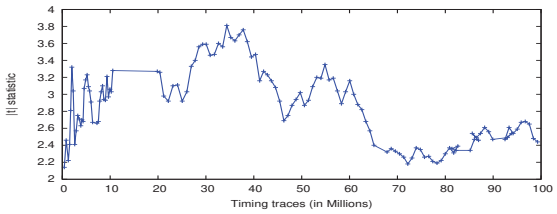

Fig. 1: Evolution of the *t*-statistic for a balanced 'if-else' branching implementation. Leaks are not detected even after 100 Million measurements

1) Analyzing Branching sequences using Dudect: We perform a simple experiment with branching sequence implementation, where we target a balanced 'if-else' structure, with statements in both branches that run in constant times. This apparently constant-time implementation, is also subjected to a fix-vs-random test, and from *Figure 1, it can be observed that even after 100 Million timing measurements, the statistical distribution of timing from the branching observations are not significantly different in order to report a leakage.*

This particular implementation is not side-channel secure, since branch misprediction traces on this implementation can lead to leaking the input on which the branching statement loops over. In such case, the Dudect timing input selection framework needs to be intelligent in order to detect the leakage from timing samples. But that defeats the purpose of non-specific statistical testing that Dudect thrives to achieve.

This example convinced us that leakage detection from timing measurements would not be efficient to design leakage detection tests for micro-architectural side-channels. *The structure of the tool must be generic which satisfies our requirement of non-specific tests, while the side-channel observations needs to be more fine-grained and precise. This motivated us to incorporate the power of precise observation from the Hardware Performance Counters*, which we elaborate in the next section.

C. Hardware Performance Counters: as a better entropy source when trying to detect micro-architectural leakages

Hardware Performance Counters (HPCs) are special purpose registers present in most of the modern-day microprocessors which store the hardware related activities during the execution of a program. The HPCs provide detailed, low-level hardware utilization statistics to advance user modules and are thus very useful in code optimization and process tuning operations.

D. Inserting HPC Measurement gadget in Dudect

We extend Dudect to an online micro-architectural analysis tool which observes the hardware event counts from Hardware Performance counters (HPCs). The measurement procedure observes the HPC traces granularly over the hardware events such as cache misprediction, branch misprediction. Since our tool is assumed to perform an online analysis, we present a practical solution based on *sampling perf event counter values* using *ioctl calls*. The idea is such that the perf object can be configured with any hardware event. The

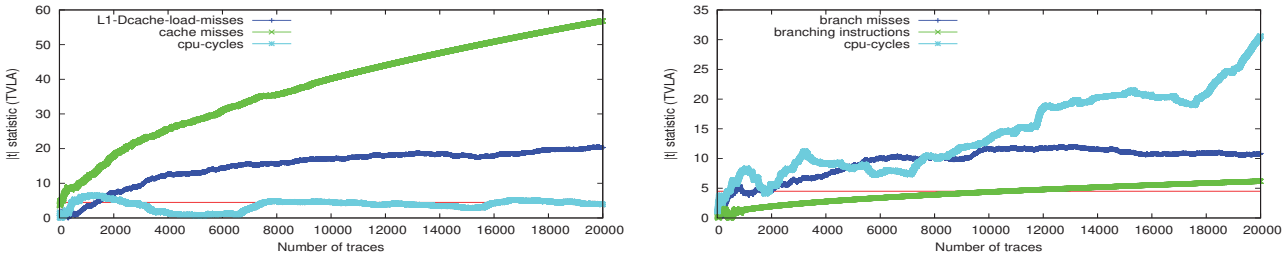

Fig. 2: TVLA values for (a) AES and (b) RSA using HPC traces over MbedTLS library

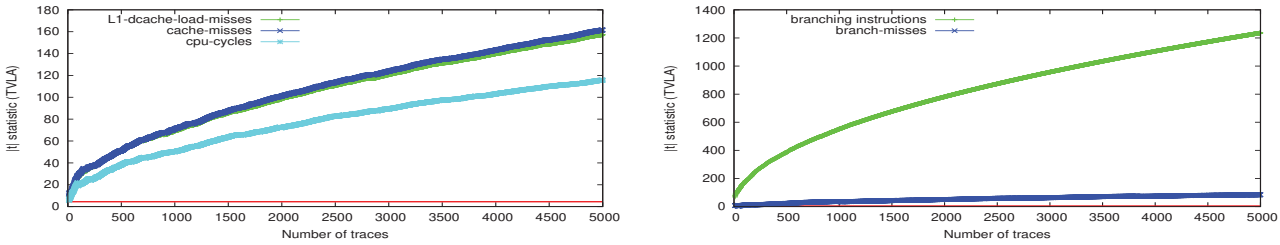

Fig. 3: `Dudect` on Spectre-V1 implementation on Ubuntu 18.04 with hardware events from HPC

full list of hardware event supported in a system is platform dependent. The *perf_event_open()*[2] snippet is incorporated to the basic structure of the open-source `Dudect`[3] to obtain seamless measurements from the HPCs.

A full list of available events can be found in various vendor's architectural manuals.[4] In this paper, we have highlighted leakage test on a few of the many events, which were interesting in context to speculative execution attacks like *L1 Data Cache load misses, cache misses, cache references, branching instructions, branch-misses, cpu-cycles and instructions*.

E. `Dudect` on cryptographic implementations

MbedTLS: We first explore the AES implementation of the MbedTLS library. We take the reference code *aescrypt2.c*[5] and plug it in our tool. The fix class corresponds to a randomly chosen input plaintext which is kept constant for all measurements, while the random class is having all random plaintext values. The key is chosen at random but kept fix within all executions. Figure 2(a) shows huge leakage, and we can see that the statistical tests reject the null hypothesis with as few as hundred measurements. The performance counters that were selected during this test were *L1-Dcache-load-misses, cache-misses* and *cpu-cycles* as reported from the perf ioctl calls.

We also tested the RSA implementation *rsa-encrypt.c* of MbedTLS, the implementation is apparently free from any obvious branching statements while the modular exponentiation is performed. But while investigating the library, the underlying *bignum.c* (which carries out the bignum operation while RSA computation happens) was implemented with extensive data-dependent branching statements, which we suspected to produce leakage when monitored over HPC event traces. Figure 2(b) shows significant leakages within a few hundred measurements. The performance counters that were selected during this test were the *number of branching instructions, branch mispredictions* and *cpu-cycles* over a fixed key, and the classes are chosen in a fix-vs-random over the input.

Following these examples, we are convinced that by incorporating the HPCs into `Dudect`, the timing leakage detection tool can be converted into a more robust framework which is also applicable in detecting micro-architectural leakages.

IV. Extending `Dudect` for Speculative Execution

A. Spectre: bounds check bypass

In this part of our experiments, we start with the first known speculative execution based attack, the Spectre-V1 (bounds check bypass). The attack works on mistraining the branch predictor. Once the branch predictor is fooled, an invalid input which is known to fail the bound-check is provided as input. The invalid input speculatively executes and fetches data from outside it memory space into the cache. The attack then thrives on a prime and probe based cache attack, which identifies the data of the unauthorized memory space.

The micro-architectural events that are responsible for success of this bound check based attack are branching and cache accesses. We have taken the reference implementation[6] as provided by the authors of the Spectre paper [4]. We divide the inputs in two classes such that, the fixed class takes a randomly chosen fixed element as input which does not have the bound check error. On the other hand, the random class is composed of inputs which are randomly chosen and mistrains the predictor followed by an input causing the bound check to fail realising the attack. Figures 3(a),(b) represent the leakages

[2]https://man7.org/linux/man-pages/man2/perf_event_open.2.html
[3]https://github.com/oreparaz/dudect
[4]https://perf.wiki.kernel.org/index.php/Tutorial
[5]https://github.com/ARMmbed/mbedtls
[6]https://gist.github.com/anonymous/99a72c9c1003f8ae0707b4927ec1bd8a

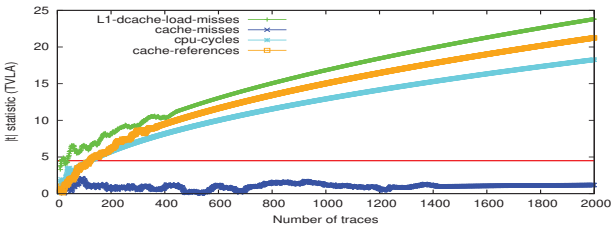

Fig. 4: `Dudect` on Meltdown implementation on Ubuntu 18.04 with hardware events from HPCs

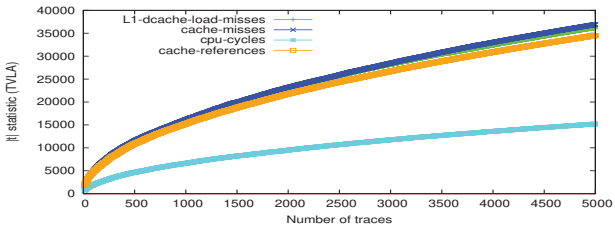

Fig. 5: `Dudect` on Foreshadow implementation on Ubuntu 18.04 with event counts from HPCs

that are detected from *L1-dcache-load-misses, cache-misses, branching instructions* event from HPC over 5000 traces. *Our tool efficiently shows the suspected leakages for stealthy cache accesses and mistraining of the branch predictor attempts of the spy process. Cache miss profile through events L1-dcache-load-misses, cache-misses leak the prime and probe accesses, while the branch misprediction leakage profile shows huge leakage because of the mistraining attempts of the spy process.*

B. Meltdown

Meltdown [5] exploits a bug in Intel's out-of-order execution engine. In out-of-order execution, the exceptions are not handled until the retirement of the instruction. Therefore, in case of an exception, the instructions are executed speculatively which create changes in micro-architectural states before getting discarded. The side channel leakage thus produced have been exploited to read data from the faulty memory access before the changes are rolled back by the processor. We have used reference implementation of the Proof of Concept (POC) published by the authors of Meltdown[7] for our experiments.

Similar to our previous experiments, the POC is plugged in our framework in a fix-vs-random testing, where the fixed set consists of a fixed input string which is read by the victim code (considered equivalent as secret). The corresponding random set consists of randomly generated strings. Figure 4 illustrates the leakages being detected over events such as *L1-dcache-load-misses, cache-references* while the overall cumulative statistics of *cache-misses* did not show any detectable leakage.

C. Foreshadow (Meltdown-P-L1)

L1 Terminal Fault (L1TF) or Foreshadow [6] show that speculative execution can leak from any virtual address where the physical address aliases with any cache line reside in the L1-Data cache. We also test the Foreshadow POC[8] with the fix-vs-random testing framework similar to Meltdown, and Figure 5 shows huge leakages being detected wrt. *L1-dcache-load-misses, cache-misses, cache-references and cpu-cycles* only in a few hundred traces.

V. Conclusion

In this paper, we propose a generic leakage detection framework tailored towards micro-architectural leakage analysis, by bringing together statistical testing with the power of granular observations from the Hardware Performance Counters. We show the applicability of the proposed tool on a wide variety of applications, starting from cryptographic applications, to popular speculative exploits and then to a handpicked examples of micro-architectural data-sampling attacks. The tool is generic, and does not require precise knowledge of the threat model, yet is able to detect leakages wrt. very specialized architectural events. The advantage of this black-box framework is that, any exploit or any executable which is suspected to be vulnerable can be plugged into the module with minimal changes to the tool, which makes it useful for generic security practitioners.

Acknowledgements

This work has been generously supported by Postdoctoral mandate KU Leuven (PDM), by the Flemish Government through FWO project TRAPS, and also funding gifted by Intel. In addition, this work is supported by the European Commission through the Horizon 2020 research and innovation programme under Cathedral ERC Advanced Grant 695305.

References

[1] 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."

[2] 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*, ser. Lecture Notes in Computer Science, T. G. "u neysu and H. Glove, Eds., vol. 9293. Springer, 2015, pp. 495–513.

[3] 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*, D. Atienza and G. D. Natale, Eds. IEEE, 2017, pp. 1697–1702. [Online]. Available: https://doi.org/10.23919/DATE.2017.7927267

[4] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[5] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[6] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds., pp. 991–1008.

[7] https://github.com/IAIK/meltdown

[8] https://github.com/vusec/ridl/tree/master/tests/source/l1tf.c