Reflections on the Experimental Evaluation of a Binary-Level Symbolic Analyzer for Spectre

Lesly-Ann Daniel
Université Paris-Saclay, CEA, List
lesly-ann.daniel@cea.fr

Sébastien Bardin
Université Paris-Saclay CEA, List
sebastien.bardin@cea.fr

Tamara Rezk
Inria
tamara.rezk@inria.fr

***Abstract*—Spectre attacks are transient execution attacks affecting modern processors and exploitable via software. Several tools have been proposed to detect Spectre vulnerabilities in software but most of these tools do not scale on real-world binary code. BINSEC/HAUNTED is one existing tool that scales and that has been evaluated on real-world binary cryptographic code.**

In this paper, we detail the experimental aspects of BINSEC/HAUNTED, but also take a step back to draw more general conclusions. We discuss general challenges and solutions regarding artifact reproducibility and availability, methodology, comparison against other tools, and pitfalls of experimental evaluation. We also discuss more specific challenges regarding Spectre vulnerability detection, e.g. prototype validation when ground truth is not easily accessible; and challenges relevant to binary-level analysis and symbolic execution, e.g. improving usability of binary-level tools or implementation choices in symbolic analyzers.

I. INTRODUCTION

Modern CPUs performance relies on complex hardware logic, including *out-of-order execution* and *speculations*. In order to improve performance, the CPU can, for instance, try to predict the next target of a conditional jump and execute instructions ahead of time. If the guess was incorrect, the CPU discards the speculative execution by reverting the affected state of the architecture. Reverted executions, also known as *transient executions*, are meant to be transparent from the architectural point of view.

Unfortunately, transient executions leave observable microarchitectural side effects that can be exploited by an attacker to recover secrets at the architectural level. This behavior is exploited in *Spectre attacks* [1] which were made public in early 2018. To date, there are four known main variants of Spectre attacks [2]. This work focuses on two of them: *Spectre-PHT* which exploits the conditional branch predictor, and *Spectre-STL* which exploits store-to-load dependencies.

Goal and challenge. In order to detect vulnerabilities to Spectre attacks in critical software—i.e. *speculative constant-time* [3] violations—we need new verification tools for low-level code which take into account the semantics of speculative execution. A well-known analysis technique that scales well on binary code is symbolic execution (SE) [4], [5]. In order to analyze speculative constant-time, it must be adapted to additionally consider transient execution introduced by the speculative semantics. However, modeling these new behaviors explicitly does not scale because it quickly leads to state explosion. Therefore, the challenge is to optimize this exploration in order to make the analysis applicable to real code.

Proposal. In our paper, "Hunting the Haunter—Efficient Relational Symbolic Execution for Spectre with Haunted RelSE" [6], we proposed a novel technique, *Haunted RelSE*, to model speculative behaviors more efficiently, and detect Spectre-PHT and Spectre-STL vulnerabilities. We implemented it in a new static analyzer for binary-code, called BINSEC/HAUNTED and evaluated it on small examples and on real-world cryptographic code. In summary, we proposed the following contributions:

- We design a dedicated technique on top of relational symbolic execution, named *Haunted RelSE*, which key idea is to model transient and sequential (in order) behavior at the same time;
- We propose a verification tool, BINSEC/HAUNTED, implementing Haunted RelSE and perform an experimental evaluation on 1) a well-known litmus tests (small test cases) for Spectre-PHT, 2) a new set of litmus tests for Spectre-STL that we propose, 3) on real-world cryptographic code. We also compare against two state of the art tools, KLEESpectre [7] and Pitchfork [3];
- Finally, we report new Spectre-STL violations concerning index-masking—a countermeasure against Spectre-PHT, and PIC options [8] from the gcc compiler.

In this paper, we detail and reflect on the experimental aspects of our work and draw general takeaways.

- Section II introduces technical details that are necessary for the comprehension of this paper. We refer the interested reader to [6] for more details on our contributions;
- Section III presents our testbed setup, along with general discussions on *availability* of artifacts and *reproducibility* of results;
- Section IV details our experimental protocol, along with general discussions on our *methodology* stemming from the software engineering community;

Learning from Authoritative Security Experiment Results (LASER) 2021
25 February 2021
ISBN 1-891562-71-1
https://dx.doi.org/10.14722/laser.2021.23286
www.ndss-symposium.org

- Section V presents the evaluation of our technique and highlights the advantage of considering *diverse metrics and use-cases* to show different facets of the performance;
- Section VI presents our comparison with state-of-the-art tools, focusing on *challenges* and illustrating the necessity to take care of *comparing underlying techniques*, not just implementations (e.g. by re-implementing the baseline);
- Section VII details the *challenges (and some solutions)* related to Spectre analysis at binary-level, combining standard challenges from binary analysis (e.g. usability) with challenges specific to Spectre (e.g. validation);
- Section VIII presents our *intermediate and unsuccessful results*, illustrating difficulties and questions that arise when implementing a symbolic analyzer;
- Section IX reports our *failures* when running or reproducing our experiments, together with solutions we devised to avoid repeating them.

The content presented in this paper is original, except for the context (Sections I and II), experimental protocol (Sections IV-A and IV-B) and experimental results (Sections V-A, V-B and VI-B), which are taken from our previous work [6] for the reader's convenience. Novel content includes a description of our testbed (Section III), discussions about our experimental setup and evaluation (Sections IV-C, V-C, VI-A and VI-C), and presentation of intermediate results and failures (Sections VIII and IX).

II. Background

This section presents necessary background on Spectre, speculative constant-time and relational symbolic execution.

Spectre attacks. In modern processors, instructions can be executed *out-of-order*, as soon as their operands are available. Processors also employ *speculation* mechanisms to predict the outcome of certain instructions before the actual result is known. Instructions streams resulting from a misspeculation—i.e. *transient executions*—are reverted at the architectural level (e.g. register values are restored) but can leave microarchitectural side effects (e.g. cache state is not restored). While these microarchitectural side effects are meant to be transparent to the program, an attacker can exploit them via side-channel attacks [9], [10]. *Spectre attacks* [1] exploit this speculation mechanism to trigger transient executions of so called *spectre gadgets* that encode secret data in the microarchitectural state, which is finally recovered via side-channel attacks. There are four variants of Spectre attacks, classified according to the speculation mechanism they exploit [2]. In this work we focus on two variants:

- Spectre-PHT [1], [11] exploits the Pattern History Table which predicts conditional branches,
- Spectre-STL [12] exploits the memory disambiguation mechanism predicting Store-To-Load dependencies.

Spectre-PHT. In Spectre-PHT, first introduced as Spectre variant 1 by Kocher et al. [1], the attacker abuses the branch predictor to intentionally misspeculate at a branch. Even if at the architectural level, a conditional statement in a program ensures that memory accesses are within fixed bounds, the attacker can lead the PHT to mispredict the value of a branch to transiently perform a memory access out-of-bounds. This out-of-bound access leaves observable effects in the cache that can ultimately be used to recover the out-of-bound read value (Listing 1).

```
uint32_t publicarray_size = 16;
uint8_t publicarray[16] = { 1 .. 16 };
uint8_t publicarray2[512 * 256];
uint8_t secretarray[16]; // Secret data
// This function encodes toLeak in the cache
void leakThis(uint8_t toLeak) {
  tmp &= publicarray2[toLeak * 512];
}
void case_1_masked(uint32_t idx) { // idx=131088
  if(idx < publicarray_size) { // Mispeculated
    // Out-of-bound read, reads secretarray[0]
    uint8_t toLeak = publicarray[idx];
    leakThis(toLeak);}} //Leaks secretarray[0]
```

Listing 1: Illustration of a Spectre-PHT attack.

Spectre-STL. To allow the CPU to transiently execute store instructions and to avoid stalling on cache-miss stores, store instructions are queued in a *store buffer*. Instead of waiting for preceding stores to be retired, a load instruction can take its value directly from a matching store in the store buffer with *store-to-load forwarding*. Additionally, when the memory disambiguator predicts that a load does not alias with pending stores, it can *speculatively bypass pending stores* in the store buffer and take its value from the main memory [13]. This behavior is exploited in the Spectre-STL [12] variant to load stale values containing secret data that are later encoded in the cache (Listing 2).

```
void case_1(uint32_t idx) {
  uint8_t* data = secretarray;
  uint8_t** data_slowptr = &data;
  (*data_slowptr)[idx] = 0; // Bypassed store
  leakThis(data[idx]);} // Leaks secretarray[idx]
```

Listing 2: Illustration of a Spectre-STL attack.

Speculative constant-time (SCT). Constant time [14] is a popular programming discipline for cryptographic code in which programs are written so that they do not store, load or branch on secret values in order to avoid leaking secrets via side-channels. However, constant-time is not sufficient to prevent Spectre attacks. For example, Listing 1 is a trivially constant-time program since there is no secret-dependent branch or memory access. However, the program is vulnerable to Spectre-PHT since an attacker can mistrain the branch predictor and leak secrets in transient execution. Speculative constant-time [3] is a recent security property that extends constant-time to take transient executions into account.

Definition 1 (Speculative constant-time [3])**.** *A program is secure w.r.t. speculative constant-time if and only if for each*

pair of (speculative) executions with the same public input and agreeing on their speculation decisions, (e.g. follow regular path or mispeculate at a branch), then their control-flow and memory accesses are equal.

Note that SCT (like constant-time and other information flow properties) is not a property of one execution trace (safety) as it relates *two execution traces* (it is a 2-hypersafety property [15]) and thus requires appropriate tools to efficiently model pairs of traces.

Symbolic execution. Symbolic Execution (SE) [4], [5], [16] consists in executing a program on *symbolic inputs*. It builds a logical formula, known as the *path predicate*, to keep track of branch conditions encountered along the execution. In order to determine if a path is feasible, the path predicate can be solved with an SMT solver [17]. SE can also check assertions in order to *find bugs* or perform *bounded-verification* (i.e., verification up to a certain depth).

Relational Symbolic Execution (RelSE). RelSE [18], [19] is a promising approach to extend SE for analyzing security properties of two execution traces such as SCT. It symbolically executes two versions of a program in the same symbolic execution instance and maximizes sharing between them. For instance, to analyze constant-time, RelSE models two programs *sharing the same public input* but with distinct secret input and, along the execution, ensures that the outcome of conditional branches and the memory indexes are equal in both execution—meaning that they do not depend on the secret.

BINSEC/HAUNTED. BINSEC/HAUNTED [6] is a binary-level analysis tool for Spectre-PHT and Spectre-STL. It relies on relational symbolic execution and performs bug-finding and bounded-verification for speculative constant-time (SCT) at binary-level. It implements an optimization called *Haunted RelSE* which key idea is to model transient and sequential behavior at the same time, while the standard approach—that we call *Explicit RelSE*—models transient executions explicitly by forking symbolic execution.

III. IMPLEMENTATION AND SETUP

This section describes the implementation of BINSEC/HAUNTED (Section III-A) and our experimental setup (Section III-B), discusses availability and reproducibility of our experiments (Section III-C), and concludes with general takeaways on reproducibility and availability of experimental evaluations (Section III-D).

A. Implementation of Binsec/Haunted

We implement our technique *Haunted RelSE* on top of BINSEC [20] binary analysis tool, and in particular we build upon its relational symbolic execution engine BINSEC/RELSE [19]. It provides:

1) a loader to easily access information encoded in the binary (ELF or PE format),
2) disassemblers from `x86`, `ARMv7` or RiscV to an intermediate representation suitable for analysis (DBA [21]),
3) a relational symbolic execution engine,
4) easy manipulation and simplification of SMT formulas and interfacing with many SMT-solvers (z3, boolector, cvc4, yices).

On top of this infrastructure, we add support for the speculative semantics and in particular, we implement our technique *Haunted RelSE* in the RelSE engine, as well as *Explicit RelSE* as a baseline, for a total of $\approx 2k$ lines of OCaml code.

B. Testbed Setup

This section details the testbed that we set up to run our experiments (experiments themselves are detailed in Section IV).

Our experiments are automated with python scripts, which set the appropriate parameters (e.g. timeout, entrypoint, initial memory, secret input, etc.) before running the analysis. Each binary is analyzed in five configurations: 1) a baseline, NoSpec, doing sequential constant-time analysis, 2) Explicit RelSE for Spectre-PHT, 3) Haunted RelSE for Spectre-PHT, 4) Explicit RelSE for Spectre-STL, 5) Haunted RelSE for Spectre-STL. Everything is pre-configured and a user just has to run the command `python expe.py` to reproduce the experiments.

Results are recorded as csv files, along with many information about the analysis (e.g. execution time, time of the SMT-solver, number of paths, number of instructions, number and type of queries sent to the solver, etc.). Finally a python script processes these csv files using the `pandas` library and generates the latex tables included in Section V.

In our experience, it is *better to record too much information than not enough*. For example, we had to rerun all our experiments because we did not record the number of unique instructions explored—which we realized later, would give a good idea of the coverage of the analysis. Moreover, diverse metrics are really useful for debugging and can give a good idea on how to improve performance (e.g. whether the bottleneck lies in the analysis or in the SMT-solver). For these reasons, we record a lot of information during our analysis (84 columns in csv files) while only a small fraction is used in the paper.

C. Availability and reproducibility

The source code of BINSEC/HAUNTED has been released on github at https://github.com/binsec/haunted and the material to reproduce the experimental evaluation has been released at https://github.com/binsec/haunted_bench, including 1) binary codes analyzed in our experiments, 2) python scripts to run BINSEC/HAUNTED on these programs, 3) csv files generated during our evaluation, 4) python script to process the results and generate latex tables, 5) instructions to reproduce experiments and run BINSEC/HAUNTED on test cases.

We also created a docker image containing the compiled code, source code, and all necessary material to reproduce the experiments—including OCaml compiler to recompile, and the boolector [22] solver, version 3.2.0, used in our experiments. We released this artifact on zenodo [23].

Going further. To be fully reproducible, our artifact should also include our evaluation of state-of-the-art tools, KLEESpectre and Pitchfork, against which we compare (see Section VI). Because the violations in binary-code depend on the compiler, we included both source code and compiled code in our artifact. To further improve reproducibility, we could also include the version of the compiler that we used to compile our use-cases (i.e. `gcc 10.1.0`). Finally, to enable other researchers to easily extend and build upon BINSEC/HAUNTED, we should add more documentation in source code.

D. Takeaways

During an experimental evaluation, it is better to record too much information than not enough, as it gives good insight on how to improve performance and can help with debugging.

Experimental setup should be designed right from the start with reproducibility and availability in mind. Because we did not think about these constraints when we first developed our experimental setup, we had to revise it entirely to automate and document our experiments and make them as easy as possible to reproduce.

Providing an artifact with tools and benchmark is the minimum for reproducible research and should be systematic when performing an experimental evaluation. Distributing source code is also very important to enable other researchers to build upon the tool, or adapt it for their comparison—as we did with Pitchfork in Section VI. For this reason, in addition to the docker artifact, we open-sourced our tool on github.

Finally, we strongly believe that security conferences should follow the example of software engineering and programming language conferences and include an artifact evaluation committee to encourage authors to share artifacts.

IV. EXPERIMENTAL PROTOCOL

This section details the experimental protocol set up in [6]. We outline clear research questions and protocol (Section IV-A), we detail the benchmarks used for our evaluation (Section IV-B) and we conclude with general takeaways on our methodology stemming from the software engineering community (Section IV-C).

A. Research questions and protocol

To assess the performance of our technique, *Haunted RelSE*, and tool, BINSEC/HAUNTED, we outline the following research questions:

RQ1 **Effectiveness.** Is our tool BINSEC/HAUNTED able to find Spectre-PHT and Spectre-STL violations in real-world cryptographic binaries?

RQ2 **Haunted vs. Explicit.** How does our technique, *Haunted RelSE* compare against the standard approach *Explicit RelSE*?

RQ3 **BINSEC/HAUNTED vs. SoA tools.** How does BINSEC/HAUNTED compare against state-of-the-art tools?

To answer RQ1, we measure the performance of BINSEC/HAUNTED on a set of real word cryptographic binaries. For RQ2, we implement the standard approach *Explicit RelSE* as a baseline in our tool and compare the performance of *Explicit* and *Haunted* explorations strategies for RelSE on a set of real word cryptographic binaries and litmus tests. Finally, for RQ3, we compare BINSEC/HAUNTED against state-of-the-art competitors, KLEESpectre [7] and Pitchfork [3].

Metrics. We evaluate performance in terms of:

- number of *unique* x86 instructions explored (I_{x86})—which gives an indication of the coverage of the analysis,
- number of paths explored (P)—which gives an indication on path explosion,
- overall execution time (T),
- number of violations (🐞), i.e. the number instructions leaking secret data,
- number of timeouts (⧗),
- number of programs proven secure (✓),
- number of programs proven insecure (✗).

These metrics give a good overview of the efficiency (I_{x86}, P, T, ⧗) and effectiveness (🐞, ✓, ✗) of the analysis.

B. Benchmark

We evaluate BINSEC/HAUNTED on the following programs:

- `litmus-pht`: 16 small insecure test cases (litmus tests) for Spectre-PHT taken from Pitchfork's modified set of Paul Kocher's litmus tests[1];
- `litmus-pht-patched`: secure litmus tests for Spectre-PHT, that we crafted by patching `litmus-pht` with index masking [24];
- `litmus-stl`: a new set of secure and insecure litmus tests that we designed for Spectre-STL[2];
- Cryptographic primitives from OpenSSL and Libsodium (detailed in Table I), including the primitives analyzed by Pitchfork [3].

This benchmark is mostly taken from Pitchfork [3] use cases, extended with secure litmus tests for Spectre-PHT and litmus tests for Spectre-STL that we mainly used for *validation*.

Programs	Type	I_{x86}	Key	Msg
`tea_encrypt`[3]	Block cipher	100	16	8
`curve25519-donna`[4]	Elliptic curve	5k	32	-
Libsodium `secretbox`[5]	Stream cipher	3k	32	256
OpenSSL `ssl3-digest-rec`[6]	HMAC	2k	32	256
OpenSSL `mee-cbc-decrypt`[6]	MEE-CBC	6k	16+32	64

Table I: Cryptographic benchmarks, with approximate static instruction count (I_{x86}) (*excluding libc code*) and sizes of secret keys and messages (Msg) in bytes.

[1] https://github.com/cdisselkoen/pitchfork/blob/master/new-testcases/spectrev1.c

[2] Open sourced at: https://github.com/binsec/haunted_bench/blob/master/src/litmus-stl/programs/spectrev4.c

[3] https://www.schneier.com/sccd/TEA.C

[4] http://code.google.com/p/curve25519-donna/

[5] https://doc.libsodium.org/secret-key_cryptography/secretbox

[6] https://github.com/imdea-software/verifying-constant-time [25]

Compilation. Programs are compiled statically for a 32-bit x86 architecture with `gcc 10.1.0`. Litmus tests, `donna` and `tea` are compiled with options `-fno-stack-protector`, `-no-pie` and `-fno-pic` in order rule out violations introduced by these options and get closer to the source-level semantics. Additionally, `donna` and `tea` are compiled for optimization levels `O0`, `O1`, `O2`, `O3`, and `Ofast`. However, Libsodium and OpenSSL are compiled in more realistic setups, with their default Makefile (including stack protector).

Challenge of stack protectors. Error-handling code introduced by stack protectors is complex and contains many system calls that cannot be analyzed directly in pure symbolic execution. BINSEC/HAUNTED stops path execution on syscalls and only jump on the error-handling code of stack protectors once per program, meaning that it might miss violations in unexplored parts of the code. Moreover, timeout is set to 1 hour for litmus tests, `tea`, and `donna`; but extended to 6 hours for code containing stack protectors (Libsodium and OpenSSL).

Setup. Experiments were performed on a laptop with an Intel(R) Xeon(R) CPU E3-1505M v6 @ 3.00GHz processor and 32GB of RAM. In the experiments, all inputs are symbolic except for the initial stack pointer `esp` (similar as related work [3]), and data structures are statically allocated. The user is expected to label secrets, all other values are public. We set the speculation depth to 200 instructions and the size of the store buffer to 20 instructions. Additionally, we only consider indirect jump targets resulting from *sequential* execution and implement a *shadow stack* to constrain return instructions to their proper return site.

C. Takeaways

Our methodology is not revolutionary, but it is not yet systematic in many security papers, while it is common in the software engineering community.

- We define clear research questions and metrics in order to make the evaluation protocol explicit and easy to understand;
- We give a particular attention to the validation of the prototype with litmus tests and cross-validation (detailed in Section VII-D);
- We compare against state-of-the-art tools in order to demonstrate progress over prior work (Section VI);
- We also implement the baseline, *Explicit RelSE*, directly in our prototype, in order to compare close implementations and truly focus on the *underlying technique* (we detail in Section VI why comparison against other tool is not sufficient).

V. PERFORMANCE OF BINSEC/HAUNTED (RQ1/RQ2)

This section presents the experimental results reported in [6] for Spectre-PHT (Section V-A) and Spectre-STL (Section V-B). Performance of BINSEC/HAUNTED are reported on a set of real word cryptographic binaries to answer RQ1; and compared to the baseline Explicit RelSE to answer RQ2. Finally, Section V-C discusses the importance of re-implementing the baseline in order to compare close implementations, and of considering diverse use cases and metrics.

A. Performance for Spectre-PHT (RQ1-RQ2)

We compare the performance of Haunted RelSE and Explicit RelSE—called Haunted and Explicit in the tables for brevity—for detecting Spectre-PHT violations. In order to focus on Spectre-PHT only, we disable support for Spectre-STL. Additionally, we also report the performance for standard constant-time verification (without speculation) as a baseline, called NoSpec. Results are presented in Table II.

Programs	PHT	I_{x86}	P	T (s)	🐞	⧗	✓	✗
`litmus-pht`	NoSpec	733	48	3	-	0	16/16	-
	Explicit	761	**703**	**10331**	**21**	**2**	-	16/16
	Haunted	761	**188**	**7**	**22**	**0**	-	16/16
`litmus-pht masked`	NoSpec	915	48	5	-	0	16/16	-
	Explicit	950	**843**	**169**	-	0	16/16	-
	Haunted	950	**182**	**8**	-	0	16/16	-
`tea`	NoSpec	326	5	.56	-	0	5/5	-
	Explicit	326	172	.62	-	0	5/5	-
	Haunted	326	172	.62	-	0	5/5	-
`donna`	NoSpec	22k	5	2948	-	0	5/5	-
	Explicit	21k	1.0M	6153	-	1	4/5	-
	Haunted	21k	1.0M	6162	-	1	4/5	-
`secretbox`	NoSpec	2721	1	5	-	0	1/1	-
	Explicit	**769**	15k	**21600**	**13**	**1**	-	1/1
	Haunted	**3583**	2.2M	**2421**	**17**	**0**	-	1/1
`ssl3-digest`	NoSpec	1809	1	4	-	0	1/1	-
	Explicit	**808**	9k	**21600**	13	**1**	-	1/1
	Haunted	**2502**	428k	**4694**	13	**0**	-	1/1
`mee-cbc`	NoSpec	6383	1	448	-	0	1/1	-
	Explicit	**696**	74k	21600	17	1	-	1/1
	Haunted	**2549**	22M	21600	17	1	-	1/1
Total	NoPHT	35k	109	3415	0	0	45/45	-
	Explicit	**25k**	1.1M	**81453**	**64**	**6**	25/25	19/19
	Haunted	**32k**	25.7M	**34892**	**69**	**2**	25/25	19/19

Table II: Performance of BINSEC/HAUNTED for Spectre-PHT.

- For `litmus-pht` and `litmus-pht-masked`, we see that Haunted RelSE: 1) explores less paths (4×) for an equivalent result, *mitigating path explosion*, 2) analyzes programs *faster* (1437× and 21× respectively), achieving performance in line with NoSpec;
- For `tea` and `donna` there is *no difference* between Explicit and Haunted. Indeed, because these programs only have a single feasible path in regular execution, Explicit RelSE forks into two paths at each conditional branch instead of four (the two other paths being unsatisfiable) which makes it equivalent to Haunted RelSE;
- Finally, for more complex programs like Libsodium and OpenSSL, Haunted RelSE achieves *better coverage* (×3.8 more instructions explored) and *less timeouts* (1/3 vs. 3/3) than Explicit RelSE.

B. Performance for Spectre-STL

In order to focus on Spectre-STL only, we disable support for Spectre-PHT. Results are presented in Table III.

The explosion of the number of paths for Explicit RelSE and its poor performance on litmus tests shows that encoding transient paths explicitly is not tractable. Haunted RelSE manages to fully explore small-size real-world cryptographic implementations (up to one hundred instructions) and to find violations in medium-size real-world cryptographic implementations (a few thousands instructions).

Overall, Haunted RelSE:

- Scales better on `litmus-stl` tests and `tea`, achieving better analysis time (3152× and 3.4× speedup respectively), producing less timeouts (0 vs. 7), and finding more violations (+24);
- While it times out on more complex code, it explores much more instruction than Explicit (8.6× more unique instructions in total), finds 126 more violations and reports 10 more insecure programs.

Programs	STL	I_{x86}	P	T (s)	🐞	⧗	✓	✗
`litmus-stl`	NoSpec	328	14	.5	-	0	14/14	-
	Explicit	316	**37M**	**7205**	13	**2**	**3/4**	10/10
	Haunted	328	**14**	**2.3**	13	**0**	**4/4**	10/10
`tea`	NoSpec	326	5	.5	-	0	5/5	-
	Explicit	278	**12M**	**18000**	**2**	**5**	-	**1/5**
	Haunted	326	**18**	**5276**	**26**	**0**	-	**5/5**
`donna`	NoSpec	22k	5	2948	-	0	5/5	-
	Explicit	**704**	12M	18000	**0**	5	-	**0/5**
	Haunted	**12k**	5	18000	**73**	5	-	**5/5**
`secretbox`	NoSpec	2721	1	5	-	0	1/1	-
	Explicit	**225**	13M	21600	**4**	1	-	**1/1**
	Haunted	**408**	2	21600	**26**	1	-	**1/1**
`ssl3-digest`	NoSpec	1809	1	4	-	0	1/1	-
	Explicit	**204**	4k	21600	**3**	1	-	1/1
	Haunted	**1763**	2	21600	**8**	1	-	1/1
`mee-cbc`	NoSpec	6383	1	448	-	0	1/1	-
	Explicit	**200**	19M	21600	**0**	1	-	**0/1**
	Haunted	**1627**	1	21600	**2**	1	-	**1/1**
Total	NoSpec	34k	27	3407	-	0	27/27	-
	Explicit	**2k**	93M	108004	**22**	**15**	**3/4**	**13/23**
	Haunted	**17k**	42	88078	**148**	**8**	**4/4**	**23/23**

Table III: Performance of BINSEC/HAUNTED for Spectre-STL.

C. Takeaways.

First, because our *use-cases are diverse* (from small litmus test to real-world cryptographic primitives), they highlight *different facets* of the performance of our analysis (especially for Spectre-PHT where we can clearly see three distinct categories with different performance). This shows that in some cases, it is preferable to present results in a non-aggregated form.

Second, some metrics (number of paths explored, execution time) are meaningful on small examples, but on larger code they become less appropriate and metrics such as number of timeouts and coverage are more meaningful, highlighting the importance of considering *diverse performance metrics*.

VI. COMPARISON AGAINST SOA TOOLS (RQ3)

To answer RQ3, we compare BINSEC/HAUNTED against two state-of-the-art tools, Pitchfork [3] and KLEESpectre [7]. This section details the challenges of such comparison (Section VI-A), the results (Section VI-B), and concludes with a general discussion on comparison with other tools (Section VI-C).

KLEESpectre [7] is an adaptation of SE for finding Spectre-PHT violations[3] (but not Spectre-STL), built on top of KLEE [26]. Pitchfork [3] is the only competing tool which can analyze programs for Spectre-STL. It is based on SE and *tainting* and implemented on top of angr [27].

A. Challenges and some solutions

Our objective is not to compare tools per se, but to compare *underlying techniques*. Unfortunately the tools are very different and many details not related to the technique can impact performance, making the comparison challenging. This section details these challenges and our approaches to mitigate them (when applicable).

LLVM vs. Binary. While BINSEC/HAUNTED and Pitchfork operate at binary-level, KLEESpectre analyzes LLVM bytecode which gives it a performance advantage. This also means that the analyzed programs are different (clang LLVM vs. gcc binaries) and might contain different vulnerabilities [19]. Nevertheless, we tried to keep the analyzed files as close as possible by providing the same compilation options.

System calls. Symbolic analyzers might process system calls differently. For instance, angr uses function summaries to model the effect of system calls on the symbolic state while BINSEC stops symbolic paths at syscalls. Because the performance of the tools eventually vary according to how they handle syscalls, we restrict this comparison to syscall-free programs `litmus-pht`, `litmus-pht-masked`, `litmus-stl`, `tea`, and `donna` and exclude `secretbox`, `ssl3-digest` and `mee-cbc`.

Reported metrics. While BINSEC/HAUNTED reports many information after its analysis (e.g. number of paths explored, number of instructions, number of queries sent to the solver, etc.), KLEESpectre and Pitchfork only report execution time and vulnerabilities found. Therefore the comparison restricts to these metrics and is less detailed than Section V.

Different properties. The properties checked by KLEESpectre, Pitchfork and BINSEC/HAUNTED are not exactly the same.

First, KLEESpectre reports several types of gadgets but only one—leak secret (LS)—can actually leak secret data and is a violation of speculative constant-time, thus we only report LS gadgets found by KLEESpectre.

Second, KLEESpectre focuses on leakage from insecure loads and does not report leakage from secret-dependent branches (missing for instance vulnerabilities using AVX-based covert channel [28]), contrary to BINSEC/HAUNTED and

[3]It also includes cache modeling—disabled for our comparison.

Pitchfork. For this reason KLEESpectre fails to report one of the litmus test as insecure (i.e. `case_10`).

Third, KLEESpectre focuses on violations during transient execution while BINSEC/HAUNTED and Pitchfork also report violation during sequential execution. Because our benchmark is constant-time in sequential execution, this does not influence the number of violations found, however this may influence the execution time as Pitchfork and BINSEC/HAUNTED have more assertions to check.

Finally, Pitchfork reports secret-dependent store in transient execution as insecure contrary to BINSEC/HAUNTED and KLEESpectre which consider them secure as they are not committed to the cache [7].

Different analysis techniques. While KLEESpectre and BINSEC/HAUNTED are based on purely symbolic relational reasoning, Pitchfork is based on standard symbolic execution with *tainting*, which is faster but possibly incorrect.

Different configurations. For Spectre-STL, Pitchfork only supports reordering loads and stores in a window of 20 instructions, and does not allow to configure the size of the store buffer. In BINSEC/HAUNTED, we can configure the speculation window (set to 200) and the size of the store buffer (set to 20)[4]. While this is the closest configuration we can get, note that a load can bypass up to *20 stores* in BINSEC/HAUNTED, which makes the window larger than *20 instructions* in Pitchfork.

Different implementation decisions. First, we had to modify Pitchfork to enable verification of Spectre-STL without Spectre-PHT (which was not possible by default).

Second, while KLEESpectre and BINSEC/HAUNTED report vulnerable instruction once, Pitchfork may report many violations at a single instruction. Thus, we post-process Pitchfork's results to report unique violations only. Note that it puts Pitchfork at a disadvantage because it still checks and reports these violations.

Third, Pitchfork stops a path after finding a violation, while BINSEC/HAUNTED continues the execution. To provide a closer comparison, we also consider a modified version of Pitchfork, namely Pitchfork-cont, which does not stop after finding a violation.

Fourth, KLEESpectre fails to report an insecure litmus test (`case_7`) for no apparent reason. We suppose that they do not consider nested speculative executions but were only able to support this hypothesis by performing few small tests.

Finally, BINSEC/HAUNTED only considers indirect jump targets resulting from sequential execution and implements a shadow stack to constrain return instructions to their proper return site. Pitchfork does not implement this mechanism and follows transient indirect jump, leading to erratic behavior such as executing non-executable sections[5]. As a consequence, it reports 6 spurious violations in non executable `.data` section.

[4]These are realistic values in modern processors.

[5]This happened in two of our Spectre-STL litmus tests.

B. Results

The results of the comparison, reported in [6], are given in Table IV.

	Programs	Tool	T (s)	⧗	🐞	✓	✗
PHT	`litmus-pht`	KLEESpectre	**1817**	0	16	2†	14/16
		Pitchfork	**1.7**	0	17	-	16/16
		Pitchfork-cont	**6.2**	0	22	-	16/16
		BINSEC/HAUNTED	**7.2**	0	22	-	16/16
	`litmus-pht masked`	KLEESpectre	**1751**	0	0	16/16	-
		Pitchfork	**10.2**	0	0	16/16	-
		Pitchfork-cont	**10.2**	0	0	16/16	-
		BINSEC/HAUNTED	**7.8**	0	0	16/16	-
	`tea`	KLEESpectre	**.4**	0	0	5/5	-
		Pitchfork	**29.5**	0	0	5/5	-
		Pitchfork-cont	**29.7**	0	0	5/5	-
		BINSEC/HAUNTED	**.6**	0	0	5/5	-
	`donna`	KLEESpectre	**7825**	**1**	0	**4/5**	-
		Pitchfork	**TO**	**5**	0	**0/5**	-
		Pitchfork-cont	**TO**	**5**	0	**0/5**	-
		BINSEC/HAUNTED	**6162**	**1**	0	**4/5**	-
STL	`litmus-stl`	Pitchfork	**21608***	**6**	11	**1/4**	**9/10**
		Pitchfork-cont	**21610***	**6**	11‡	**1/4**	**9/10**
		BINSEC/HAUNTED	**2.3**	**0**	13	**4/4**	**10/10**
	`tea`	Pitchfork	**TO**	**5**	**0**	-	**0/5**
		Pitchfork-cont	**TO**	**5**	**0**	-	**0/5**
		BINSEC/HAUNTED	**5275**	**0**	**26**	-	**5/5**
	`donna`	Pitchfork	TO	5	**0**	-	**0/5**
		Pitchfork-cont	TO	5	**0**	-	**0/5**
		BINSEC/HAUNTED	TO	5	**73**	-	**5/5**

Table IV: Performance of BINSEC/HAUNTED, Pitchfork and KLEESpectre on `tea`, and Spectre-PHT and Spectre-STL litmus tests. Timeout (⧗) is set to 1 hour. †False positives. ‡Excluding 6 spurious violations in (non executable) `.data` section. *Excluding ⧗, times are respectively 8.1 and 10.6.

For Spectre-PHT, KLEESpectre, as expected, shows similar trend as Explicit RelSE in Table II. It is slightly faster than BINSEC/HAUNTED on `tea` (1.5×), but slower on `litmus-pht` (250×) on `litmus-pht-masked` (224×). On the contrary, Pitchfork does not seem to follow an Explicit exploration strategy as it scales well on litmus tests. Pitchfork-cont is slightly faster than BINSEC/HAUNTED (1.2×) on `litmus-pht`, but it is 50× slower on `tea` and times-out on `donna`.

For Spectre-STL however, Pitchfork follows the explicit strategy which quickly leads to state explosion, poorer performance and more timeouts. The analysis runs out-of-memory—taking 32GB of RAM—for six cases of `litmus-stl`, 1 `tea`, and 4 `donna`. Hence, Pitchfork does not scale for Spectre-STL even on small-size binaries whereas BINSEC/HAUNTED can exhaustively explore small-size binaries. Our results further show that BINSEC/HAUNTED finds 112 more Spectre-STL violations, identifies 11 more insecure programs and establishes security of 3 more programs compared to Pitchfork.

C. Takeaways

Comparing different tools is challenging as performance eventually depends on implementation details and might not

reflect what we really want to measure—the underlying technique. Consequently such comparison must be taken with a pinch of salt.

For this reason, we believe that implementing our own *Explicit RelSE* baseline inside BINSEC/HAUNTED to compare against *Haunted RelSE* is a good solution, allowing to compare very close implementations and focus on the underlying technique (Section V).

In our experiments, we always tried our best not to put KLEESpectre and Pitchfork at a disadvantage (e.g. larger load reordering window in BINSEC/HAUNTED than in Pitchfork). However it was not always easy or possible—e.g. different performance due to different implementation decisions are hard to mitigate.

Finally, we did not encounter any difficulty to run Pitchfork and KLEESpectre on our own test cases, and adapting the implementation of Pitchfork for our comparison was quite simple, which is truly appreciable.

VII. CHALLENGES: BINARY-LEVEL & SPECTRE ANALYSIS

Implementing a verification tool for Spectre at binary-level poses a combination of challenges: standard challenges of binary-level (Section VII-A) and information-flow analysis (Section VII-B), difficulties to interpret the results (Section VII-C), and validation of the tool (Section VII-D). This section presents some of the challenges we faced with BINSEC/HAUNTED, the solutions we adopted, some opportunities for improvement and concludes with general advice towards improving binary-level analyzers (Section VII-E).

A. Standard challenges in binary-level analysis

Configuring initial memory. In binary level symbolic execution, the initial memory is symbolic by default, but some parts must be initialized with information from the binary. For instance sections `.data` and `.rodata` contain initialized data, and a load from one of these sections should read data directly from the binary. However, the case of the `.bss` section is trickier as it contains both variables initialized to 0 — that we would like to set to 0 — and uninitialized variables — that we would like to keep symbolic. Our solution is to keep the `.bss` section symbolic by default and, when necessary, to do some *reverse engineering* to specify the address ranges to initialize to 0.

Deal with indirect functions. Indirect functions [29] are functions whose implementation is chosen at runtime, using a resolver function. They are used in the GNU standard library (glibc) to implement for instance multiple version of `memset` chosen depending on the CPU. To avoid analyzing multiple implementations of `memset`, we replace calls to the resolver function `__memset_ifunc` to calls to the specific implementation `__memset_ia32`.

Limitations of BINSEC/HAUNTED. BINSEC symbolic execution engine does not have function summaries (a.k.a. stubs) for the standard library or system calls. Therefore, we only apply BINSEC/HAUNTED to statically compiled binaries and stop a path when encountering a syscall[6]. BINSEC does not easily handle dynamically allocated memory, thus we statically allocate buffers.

B. Specifying secrets

BINSEC/HAUNTED has three different approaches for specifying secrets, offering different trade-offs between usability and realism.

1) Reverse engineering. A first approach to specify secret input is to specify them as offsets from the initial stack pointer `esp`. This approach requires manual *reverse engineering* to identify secret and compute their offsets (see Fig. 1).

Pro: Realistic and does not require any change in source code.

Con: Requires manual analysis and must be done each time the program is recompiled. Therefore it is not appropriate for large-scale experiments or for testing multiple compilers on the same program.

```
out= dword ptr -20h
data= dword ptr -18h
key= dword ptr -10h

push    ebp
mov     ebp, esp
sub     esp, 20h
lea     eax, [ebp+key]
push    eax
lea     eax, [ebp+out]
push    eax
lea     eax, [ebp+data]
push    eax
call    encipher
```

Figure 1: Binary disassembled with IDA. Let us call esp_0 the initial value of the stack pointer. Note that the `push` instruction increments `esp` by 4, thus `ebp` is set to esp_0+4. From there we can compute that symbolic secret input `key`, `data`, `out` are located at offsets `0x10+0x4`, `0x18+0x4`, `0x20+0x4` of esp_0.

2) Use function stubs. A second approach to specify secrets directly in the source code is to use dummy functions as illustrated in Listing 3. In symbolic execution, a call to `high_input_16(key)` is replaced by a function summary that initializes the memory at address `key` with 16 symbolic byte considered as secret.

Pro: Does not require reverse-engineering and automatically applies to any binary compiled from the source, making it suitable for large-scale experiments or for testing multiple compilers.

Con: Requires to either modify the source code or to put a wrapper around the code to analyze (e.g. around the call to a library as illustrated in Listing 3). Dummy function calls insert loads and stores which can introduce additional Spectre-STL violations, therefore this approach might not be ideal for studying Spectre-STL.

3) Use global variables. A global variable in the source program is identified in the binary with a symbol that contains its name and address. In BINSEC/HAUNTED, a user can specify which global variables contain secret input, and BINSEC/HAUNTED will take care of initializing the corresponding

[6]This only happened in the error-handling code of the stack protectors.

```
// Declare symbolic secret input
uint8_t key[16]; high_input_16(key);
uint8_t data[8]; high_input_8(data);
uint8_t out[8]; high_input_8(out);

// Function to analyze
encipher(data, out, key);
```

Listing 3: Specify secrets with dummy functions in C source.

addresses with symbolic secret values. This is the approach used in our experimental evaluation.

Pro: Simple, automatic, and avoids introducing new STL-violations.

Con: Not very realistic as secret data would not be stored directly in the binary as global variables.

Takeaway: specification at binary-level. Specifying security policies at binary level is more challenging as developers have to transpose their reasoning from source code to binary code (e.g. from program variables to memory addresses). Instrumentation at source level can improve automation and usability—which are necessary to run large scale experiments—but is not always realistic, or even possible. In BINSEC/HAUNTED, we propose different approaches for specifying secrets that offer different trade-offs between usability and realism.

C. Facilitate interpretation of counterexamples

Improving usability of Spectre-detection tools—in particular the interpretation of counterexamples—is crucial in order to facilitate their validation. This section, details the strategies implemented in BINSEC/HAUNTED in order to facilitate the interpretation of counterexamples and highlights potential opportunities for improvement.

Counterexample returned by BINSEC/HAUNTED. When detecting a violation, BINSEC/HAUNTED returns 1) the instruction that leaks secrets and its location, 2) the initial configuration (memory and registers) that trigger the violation. We also implemented an IDA script to visualize the coverage of the analysis and highlight the violations found, allowing a user to directly identify the instruction triggering the violation in the assembly code.

Spectre-STL. For Spectre-STL, BINSEC/HAUNTED must additionally return the interleaving of loads and stores leading to the violation. Because the choice of loads and stores interleaving is encoded in boolean variables and left to the solver [6], we have to encode this information in the formula and extract if from the model returned by the solver. To do this, we encode the address of loads and address of stores in the name of the boolean variables that encode the choices of the solver. For instance, if the solver sets the boolean variable `load_08049d1c_from_08049cf5` to true, it means that the load at address `0x08049d1c` takes its value from the store at address `0x08049cf5`. Similarly, if the variable `load_08049d27_from_main-mem` is set to true, the load instruction at address `0x08049d27` takes its value from the initial memory.

Further improving usability. In the initial memory configuration returned by BINSEC/HAUNTED as a counterexample, the user has to make the link between memory addresses and variables in the source code. Even though the reverse-engineering task is not difficult, it would enhance usability to automatically link memory locations to program variables when possible e.g. using symbols for global variables; or giving the user the possibility to specify local variables of interest at the source level.

BINSEC/HAUNTED could also easily improve the quality of counterexamples for Spectre-PHT by reporting information on the source of speculation—e.g. the address of the mispeculated conditionals, like in KLEESpectre.

Finally, another improvement would be to differentiate whether a SCT violation occurs in sequential or in transient execution[7], as these two types of violations require different countermeasures.

D. Validation of BINSEC/HAUNTED

Validating results from BINSEC/HAUNTED is challenging as there is no ground truth (especially for Spectre-STL), and SCT violations are difficult to find manually.

Litmus for Spectre-PHT. To validate BINSEC/HAUNTED for Spectre-PHT, we mainly used the set of litmus tests developed by Paul Kocher [30] (precisely, the modified version from Pitchfork's benchmark[8]) which is a set of 16 insecure simple test cases developed to test mitigations introduced by compilers. However, it still required manual analysis to precisely identify violations (e.g. number of vulnerabilities, locations, etc.). Additionally, we created a new set of secure litmus tests by applying the index-masking countermeasure to this initial set of litmus tests for Spectre-PHT.

Litmus for Spectre-STL. Validating BINSEC/HAUNTED for Spectre-STL was more challenging as there is no ground truth except for the initial proof-of-concept[9]. Moreover it is even more difficult to manually identify (or even confirm) vulnerabilities as it requires to reason about different load and store interleavings. Therefore, to validate BINSEC/HAUNTED, we manually crafted and documented 14 litmus tests for Spectre-STL[10].

Cross validation. For both Spectre-PHT and Spectre-STL, we compared BINSEC/HAUNTED against Pitchfork and KLEESpectre on these litmus test (when possible) and manually checked the results in case of deviation. Finally, we also use the test cases for regression testing for BINSEC/HAUNTED.

[7] Recall that SCT [3] prevent leaks in both sequential and transient execution

[8] https://github.com/cdisselkoen/pitchfork/blob/master/new-testcases/spectrev1.c

[9] https://github.com/IAIK/transientfail/tree/master/pocs/spectre/STL

[10] Open sourced at https://github.com/binsec/haunted_bench/blob/master/src/litmus-stl/programs/spectrev4.c

Connection to real Spectre attacks. Finally, making the connection between real attacks and SCT violations discovered by the tools is an open question. Determining if SCT violations are exploitable requires a good understanding of the details of the micro-architecture and is micro-architecture-specific. A step to get closer to real attacks could be to extend SCT with details on the micro-architecture (e.g. adding conditions under which a processor may speculate).

Takeaway: validation of Spectre analyzers. The lack of ground truth and the intricate nature of Spectre vulnerabilities (i.e. combination of speculation and side channels) makes it difficult to validate analyzers.

However, we believe that it would be difficult to provide a *precise and generic* benchmark for validation. Should such a benchmark be provided at source-level, the vulnerabilities in the binary-code would vary with compilation, and thus the benchmark would be imprecise; should it be provided at binary-level for a specific architecture, it would exclude LLVM-level tools, tools that do not handle the specific architecture, and tools that need to recompile from source to instrument the binary.

One way to help with validation is to improve the *usability* of the tools, in order to make interpretation of the results and cross validation easier.

E. Takeaways

Binary-level reasoning make many things more challenging, e.g. interpretation of the results, as the user has to transpose their reasoning from source code to binary code, and even cross-validation as other tools do not necessarily support the same architecture.

Most specification tasks requiring reverse engineering—like configuring the initial memory or specifying secrets—can be partially automated with source-level instrumentation (e.g. using dummy functions), or using symbol information.

We believe that *improving usability and automation* is one of the keys to overcome these challenges. Indeed, usability and automation are crucial to run large scale experiments but also to be able to understand counterexamples returned by the analysis and can greatly help in prototype validation.

VIII. Loose and unsuccessful results

When developing a symbolic analyzer like BINSEC/HAUNTED there are many implementation choices to make, especially on how to build the formula sent to the solver. Section VIII-A illustrates some implementation choices that we faced, Section VIII-A reports our unsuccessful results, and Section VIII-C summarizes the lessons we learned.

Note the caveat that these experiments were not rigorously evaluated nor documented, thus they must be taken not as facts but as illustrations of the kind of questions that arise when implementing a symbolic analyzer.

A. Loose intermediate results

Choosing the solver. BINSEC can interface with many SMT solvers: boolector, z3, yices, cvc4. Boolector is currently the best on the theory used in our symbolic execution (i.e. QF_ABV) [31], [32]. Nevertheless, we still ran some experiments and confirm that it was indeed better than the others in our case. However, on some programs, boolector failed to solve the constraint while z3 was able to quickly find a solution.

Specifying path constraint. In symbolic execution the path constraint is the conjunction of conditional expressions encountered along the execution. In the SMT-LIB [33] format, there are different solutions to specify this constraint. We experimented on two variants: 1) put assertions in the formula straightaway after the conditional instructions, 2) put all conditions in a single conjunction at the end of the formula. In our experiments, it did not make any difference so we chose the first variant as it makes the formula more readable.

Simpler formula $\neq$ faster to solve. In RelSE for (speculative) constant-time, when encountering a conditional statement `if` $\subset$ where $\subset$ maps to a pair of expressions $\langle c_l \mid c_r \rangle$, we first send an *insecurity query* to make sure that the evaluation of the condition is the same in both execution—i.e. that $(c_l = 0) \neq (c_r = 0)$ is unsatisfiable (meaning that $(c_l = 0)$ if and only if $(c_r = 0)$). Then, we continue the execution along both branches and add the corresponding condition to the path constraint, e.g. $pc \wedge c_l = 0 \wedge c_r = 0$ to follow the *else* branch. Note that because we know from the previous insecurity query that $(c_l = 0)$ equals $(c_r = 0)$), we can simplify the path constraint as $pc \wedge c_l = 0$. While this path constraint is simpler, our experiments showed that it was actually slower to solve.

B. Unsuccessful results

Trying to help solver. In RelSE for (speculative) constant-time, *a lot of insecurity checks* are added to ensure that memory indexes and conditional expressions do not depend on secret. We noticed that most of these checks were redundant (especially when compiling without optimizations) and added an optimization to remove duplicated checks. While this optimization significantly reduced the size of the formula (usually around 30%, up to 50%), it had not impact on the solving time of the formula.

Propagate information in symbolic state. In RelSE for Spectre-STL, a variable v maps to a set of symbolic expressions $\mathrm{v} \mapsto \{a, b, c, d\}$, corresponding to the sequential and transient values (resulting from different load and store interleavings). When a transient value is retired, for instance a, we have to add this information to the SMT formula, call it φ. Without entering into unnecessary details, we can do this by just adding an assertion to the formula i.e. similarly as $\varphi \wedge \mathrm{v} \neq a$. Note that this is sufficient but not optimal, as in the symbolic state we still have $\mathrm{v} \mapsto \{a, b, c, d\}$, potentially preventing extra simplifications. We implemented an optimization to propagate the information to the symbolic state to effectively get $\mathrm{v} \mapsto \{b, c, d\}$. Unfortunately, this optimizations did not improve the performance of our analysis.

C. Takeaways.

These intermediate results were obtained from superficial evaluations and, retrospectively, we should have documented them more carefully. *Even intermediate and unsuccessful experiments should be performed rigorously and documented carefully*, so their results can also be useful to others, durably support implementation decisions, and not be forgotten.

In our experience *SMT-solver can be hard to satisfy*, and it can be frustrating to spend time implementing an optimization which is not efficient the end. From our experience with solvers, trivial formula simplifications are quite unlikely to improve performance.

IX. Failures with experiments

In this section, we report our failures when running or reproducing our experiments and the solutions we devised to avoid repeating them, in the hope that it helps others avoiding these pitfalls.

Deal with out-of-memory. State-explosion in Pitchfork for Spectre-STL leads to high memory usage and eventually runs out-of-memory—taking 32GB of RAM and crashing the user session, preventing from running additional experiments. To be able to deal with out-of-memory while running experiments, we used earlyoom [34], a daemon which kills processes when the amount of available memory and swap is too low. We first tried not to be too conservative and set the swap threshold to 50% but most of the time it failed to kill Pitchfork, and crashed our experiments. In the end, we adopted a more conservative approach and configured it to kill Pitchfork when the memory consumption reaches 90% and the swap reaches 5%.

Beware other programs running. Having several programs running on a machine can be a problem when running experiments, especially on shared architectures, where a user does not have full control on which programs are running. For instance, we started to run our experiments on a shared sever but quickly noticed erratic results. Then, we realized the server was sometime running some other programs, taking up a lot of ressource and impacting our performance. The best solution is to have a dedicated machine to run experiments but it is not always possible. In our case, we run experiments on a laptop when not in use—during the night or week-end—after rebooting.

Minor version changes can have big impact. When trying to reproduce some of our own results, we noticed some significant slowdowns. After (time-consuming) investigation, we finally identified the cause of the performance degradation as a boolector upgrade from 3.2.0 to 3.2.1, leading to a significant increase of memory consumption.

Don't forget check list. From our failures, we also derived a small checklist that we check before running our experiments:

- Plug the computer before running experiments—we recorded -50% performance on battery;
- Disable auto-suspend—or you computer might just go to sleep in the middle of you experiments;
- Check that the frequency of the CPU is not stuck at 800MHz instead of 4GHz—yes, that happened to us and we had a hard time finding why our tool was $4\times$ slower than usual, so now, before running our experiments, we check `grep "cpu MHz"/proc/cpuinfo`.

Takeaways

These examples of failures illustrate that it is not trivial to run quality experiments and to reproduce results. In our experience, failures to reproduce experiments can have many possible causes and are often time-consuming to debug. In order to spot problems early, we encourage researchers to run regression tests (or try to reproduce some of their results) to spot performance changes (e.g. after modifying the implementation, or upgrading a dependency). Recording the commit-hash of the implementation during experiments can also help reproducing results and reducing debugging effort.

X. General takeaways

Artifact sharing is necessary for reproducible research but it also greatly helps the progress of research, allowing other researchers to reuse benchmarks, to compare their work, to build upon the artifacts, etc. For these reasons, we believe that security conferences should include an artifact evaluation track to encourage authors to share their artifacts.

We also encourage authors to follow a rigorous methodology in their experimental protocol, setting clear research questions, taking care of comparing underlying techniques and not just implementations (e.g. by re-implementing the baseline), etc.

Regarding binary-level analysis, we argue that one of the keys to improve analyzers is to improve usability and automation. Indeed, they are crucial to run large scale experiments, to understand the results of the analysis, and most importantly can greatly help in prototype validation.

We also argue that even intermediate and unsuccessful experiments should be performed rigorously and documented carefully, so they can also be more useful to the researcher but also to others, durably support implementation decisions, and not be forgotten.

Finally, it is not trivial to run quality experiments and to reproduce results. Failures to reproduce experiments can have many possible causes and are often time-consuming to debug. We do not have a universal solution but encourage researchers to run regression tests and try to reproduce their experiments to spot problems early.

Acknowledgment

We warmly thank the organizers of the LASER workshop for giving us the opportunity to develop the experimental part of our work in greater depth. We also thank the participants of the workshop for the inspiring discussions.

References

[1] 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 *IEEE Symposium on Security and Privacy*, 2019.

[2] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses", in *USENIX Security Symposium*, 2019.

[3] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era", in *PLDI*, 2020.

[4] J. C. King, "Symbolic execution and program testing", *Communications of the ACM*, 1976.

[5] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: Whitebox fuzzing for security testing", *Communications of the ACM*, 2012.

[6] L.-A. Daniel, S. Bardin, and T. Rezk, "Hunting the Haunter—Efficient relational symbolic execution for spectre with haunted RelSE", in *NDSS*, 2021.

[7] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury, "KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution", *ACM Trans. Softw. Eng. Methodol.*, 2020.

[8] F. S. Foundation. "Position Independent Code (GNU Compiler Collection (GCC) Internals)", [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/PIC.html (visited on 10/12/2020).

[9] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems", in *Annual International Cryptology Conference*, 1996.

[10] D. J. Bernstein, "Cache-timing attacks on AES", 2005.

[11] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses", *CoRR*, 2018.

[12] J. Horn. "Speculative execution, variant 4: Speculative store bypass". (2018), [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 (visited on 10/12/2020).

[13] "Intel® 64 and IA-32 Architectures Optimization Reference Manual", Intel, [Online]. Available: https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html (visited on 10/12/2020).

[14] G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: The case of cryptographic "Constant-Time"", in *CSF*, 2018.

[15] M. R. Clarkson and F. B. Schneider, "Hyperproperties", *Journal of Computer Security*, 20, 2010.

[16] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later", *Communications of the ACM*, 2013.

[17] J. Vanegue and S. Heelan, "SMT solvers in software security", in *WOOT*, 2012.

[18] G. P. Farina, S. Chong, and M. Gaboardi, "Relational symbolic execution", in *PPDP*, 2019.

[19] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level", in *IEEE Symposium on Security and Privacy*, 2020.

[20] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion, "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis", in *SANER*, 2016.

[21] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The BINCOA framework for binary code analysis", in *CAV*, 2011.

[22] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0 system description", *Journal on Satisfiability, Boolean Modeling and Computation*, 2014 (published 2015).

[23] L.-A. Daniel, S. Bardin, and T. Rezk, *Docker image of Binsec/Haunted tool*, version 1.0, 2021. DOI: 10.5281/zenodo.4442337. [Online]. Available: https://doi.org/10.5281/zenodo.4442337.

[24] F.Pizlo, "What spectre and meltdown mean for WebKit", 2018. [Online]. Available: https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/ (visited on 07/17/2020).

[25] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC", in *FSE*, 2016.

[26] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.", in *OSDI*, 2008.

[27] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis", in *IEEE Symposium on Security and Privacy*, 2016.

[28] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read arbitrary memory over network", in *ESORICS (1)*, 2019.

[29] "The GNU indirect function support (IFUNC)", [Online]. Available: https://sourceware.org/glibc/wiki/GNU_IFUNC (visited on 04/23/2021).

[30] P. Kocher. "Spectre Mitigations in Microsoft's C/C++ Compiler". (13, 2018), [Online]. Available: https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html (visited on 07/30/2020).

[31] "SMT-COMP", SMT-COMP, [Online]. Available: https://smt-comp.github.io/2019/results.html (visited on 10/11/2019).

[32] B. Farinier, R. David, S. Bardin, and M. Lemerre, "Arrays made simpler: An efficient, scalable and thorough preprocessing", in *LPAR*, 2018.

[33] C. Barrett, P. Fontaine, and C. Tinelli, “The SMT-LIB Standard: Version 2.6”, 2017.

[34] “Earlyoom—the early oom daemon”, [Online]. Available: https://github.com/rfjakob/earlyoom (visited on 04/20/2021).