Hunting the Haunter — Efficient Relational Symbolic Execution for Spectre with Haunted RelSE

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 are microarchitectural attacks which were made public in January 2018. They allow an attacker to recover secrets by exploiting speculations. Detection of Spectre is particularly important for cryptographic libraries and defenses at the software level have been proposed. Yet, defenses correctness and Spectre detection pose challenges due on one hand to the explosion of the exploration space induced by speculative paths, and on the other hand to the introduction of new Spectre vulnerabilities at different compilation stages. We propose an optimization, coined *Haunted RelSE*, that allows scalable detection of Spectre vulnerabilities at binary level. We prove the optimization semantically correct w.r.t. the more naive explicit speculative exploration approach used in state-of-the-art tools. We implement Haunted RelSE in a symbolic analysis tool, and extensively test it on a well-known litmus testset for Spectre-PHT, and on a new litmus testset for Spectre-STL, which we propose. Our technique finds more violations and scales better than state-of-the-art techniques and tools, analyzing real-world cryptographic libraries and finding new violations. Thanks to our tool, we discover that index-masking—a standard defense for Spectre-PHT—and well-known gcc options to compile position independent executables introduce Spectre-STL violations. We propose and verify a correction to index-masking to avoid the problem.**

I. Introduction

Modern CPUs performance relies on complex hardware logic, including branch predictors and *speculations*. Independently from the hardware implementation, the architecture describes how instructions behave in a CPU and includes state that can be observed by the developer such as data in registers and main memory. The microarchitecture describes how the architecture is implemented in a processor hardware, and its state includes for example entries in the cache which are transparent to the developer. In order to reduce clock cycles, the CPU can execute instructions ahead of time, and attempt, for instance, to guess values via a branch predictor to speculatively execute a direction of the control flow. If the guess was incorrect, the CPU discards the speculative execution by reverting the affected state of the architecture. At the end, only correct executions will define the state of the architecture. Reverted executions, also known as *transient executions*, are meant to be transparent from the architectural point of view.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24286
www.ndss-symposium.org

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. Since then, Spectre attacks have drawn considerable attention from both industry and academy, with works that discovered new Spectre variants [2], new detection methods [3]–[5], and new countermesures [6]–[8]. To date, there are four known main variants of Spectre attacks [2]. Most works on analyzers [3], [4], [9]–[12] only focus on the Pattern History Table (PHT) variant (a.k.a Spectre-v1 [1]) which exploits conditional branches, yet they struggle on medium-size binary code (cf. Table V). Only one tool, Pitchfork [5], addresses the Store to Load (STL) variant (a.k.a Spectre-v4 [13]), which exploits the memory dependence predictor. Unfortunately, Pitchfork does not scale for analyzing Spectre-STL, even on small programs (cf. Table IV). Other variants are currently out-of-scope of static analyzers (see Sections II and VII).

Goal and challenges. In this paper, we propose a novel technique to detect Spectre-PHT and Spectre-STL vulnerabilities and we implement it in a new static analyzer for binary code. Two challenges arise in the design of such an analyzer:

- **C1** First, the details of the microarchitecture cannot be fully included in the analysis because they are not public in general and not easy to obtain. Yet the challenge is to find an abstraction powerful enough to capture side channels attacks due to microarchitectural state.
- **C2** Second, exploration of all possible speculative executions does not scale because it quickly leads to state explosion. The challenge is how to optimize this exploration in order to make the analysis applicable to real code.

Proposal. We tackle challenge C1 by targeting a relational security property coined in the literature as *speculative constant-time* [5], a property reminiscent of constant-time [14], widely used in cryptographic implementations. Speculative constant-time takes speculative executions into account without explicitly modeling intrincate microarchitectural details. However, it is well known that constant-time programming is not necessarily preserved by compilers [15], [16], so our analysis operates at binary level—besides, it is compiler-agnostic and does not require source code. For this, we extend the model of previous work for binary analysis of constant-time [16] in order to analyze *speculative constant-time* [5].

A well-known analysis technique that scales well on binary code is symbolic execution (SE) [17], [18]. However, in order

to analyze speculative constant-time, it must be adapted to consider the speculative behavior of the program. Symbolic analyzers for Spectre-PHT [3], [4], [12] and Spectre-STL [5] model the speculative behavior *explicitly*, by forking the execution to explore transient paths, which quickly leads to state explosion—especially for Spectre-STL which has to fork for each possible store and load interleaving. The adaptation of symbolic execution to constant-time-like properties, known as *relational symbolic execution* (RelSE), has proven very successful in terms of scalability and precision for binary level [16]. In order to address C2, our key technical insight is to adapt RelSE to execute transient executions *at the same time* as regular executions (i.e. executions related to correct speculations). We name this technique *Haunted RelSE*:

- For Spectre-PHT, it prunes redundant states by executing at the same time transient and regular paths resulting from a conditional statement;
- For Spectre-STL, instead of forking the symbolic execution for each possible load and store interleaving, it prunes redundant cases and encodes the remaining ones in a single symbolic path.

We implement Haunted RelSE in a relational symbolic analysis tool called BINSEC/HAUNTED for binary level. For evaluation, we use the well-known Kocher test cases for Spectre-PHT [1], as well as a *new set of test cases that we propose for Spectre-STL*, and real-world cryptographic code from donna, Libsodium and OpenSSL libraries.

Findings. Interestingly, our experiments revealed that index-masking [19], a well-known defense used against Spectre-PHT in WebKit for example, *may introduce new Spectre-STL vulnerabilities*. We propose and verify safe implementations to deal with this problem. By means of our tool, we have also discovered that a popular option [20] of gcc to generate position-independent code (PIC) *may introduce Spectre-STL vulnerabilities*. We also confirm, as already reported by Cauligi et al. [5], that the stack protections added by compilers introduce Spectre violations in cryptographic primitives.

Contributions. In summary, our contributions are:

- We design a dedicated technique on top of relational symbolic execution, named Haunted RelSE, to efficiently analyze speculative executions in symbolic analysis to detect PHT and STL Spectre violations (Sections III and IV). *The main idea behind Haunted RelSE is to symbolically reason on regular and transient behaviours at the same time*. Even though our encoding for memory speculations is reminiscent of some encodings for state merging [21]–[23], we actually follow different philosophy, by preventing artificial splits between regular and transient executions rather than trying to pack together different (possibly unrelated) paths. We formally prove that relational analyses modeling all speculative executions explicitly, or using Haunted RelSE are semantically equivalent (Section IV);
- We propose a verification tool, called BINSEC/HAUNTED, implementing Haunted RelSE and evaluate it on well-known litmus tests (small test cases) for Spectre-PHT. We further propose a new set of litmus tests for Spectre-STL as a contribution and test BINSEC/HAUNTED on it. Experimental evaluation (Section V) shows that BINSEC/HAUNTED can find violations of speculative constant-time in real-world cryptographic code, such as donna, Libsodium and OpenSSL libraries. For Spectre-PHT, BINSEC/HAUNTED can exhaustively analyze code up to 5k static instructions. It is faster than the (less precise) state of the art tools KLEESpectre and Pitchfork. For Spectre-STL, it can exhaustively analyze code up to 100 instructions and find vulnerabilities in code up to 6k instructions; compared to state-of-the art tool Pitchfork, BINSEC/HAUNTED is significantly faster, finds more vulnerabilities, and report more insecure programs.
- To the best of our knowledge, we are the first to report that the well-known defense against Spectre-PHT, index-masking, may introduce Spectre-STL vulnerabilities. We propose correct implementations, verified with our tool, to remedy this problem (Section VI). We are also the first to report that PIC options [20] from the gcc compiler introduce Spectre-STL violations (Section VI).

Discussion. While Spectre attacks opened a new battlefield of system security, reasoning about speculative executions is hard and tedious. There is a need for automated search techniques, yet prior proposals suffer from scalability issues due to the path explosion induced by extra speculative behaviors. Haunted RelSE is a step toward scalable analysis of Spectre attacks. For Spectre-PHT, Haunted RelSE can dramatically speed up the analysis in some cases, pruning the complexity of analyzing speculative semantics, and scales on medium-size real-world cryptographic binaries. For Spectre-STL, it is the first tool able to exhaustively analyze small real world cryptographic binaries and find vulnerabilities in medium-size real world cryptographic binaries.

II. BACKGROUND

We provide here basic background on Spectre, speculative constant-time and relational symbolic execution.

Spectre attacks. In modern processors, instructions are fetched in order and placed in a *reorder buffer* where instructions can be executed in any 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 mispeculation—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 [24], [25]. *Spectre attacks* [26] 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]:

- Spectre-PHT [26], [27] exploits the Pattern History Table which predicts conditional branches,
- Spectre-BTB [26] exploits the Branch Target Buffer which predicts branch addresses,

- Spectre-RSB [28], [29] exploits the Return Stack Buffer which predicts return addresses,
- Spectre-STL [13] exploits the memory disambiguation mechanism predicting Store-To-Load dependencies.

Speculation mechanisms at the root of BTB and RSB variants can, in principle, be mistrained to jump to arbitrary addresses [5], [29], which seems to be intractable for static analyzers (cf. Section VII). For this reason, we focus in this paper on Spectre-PHT and Spectre-STL variants.

Spectre-PHT. At the microarchitectural level, the Pattern History Table (PHT) predicts the outcome of conditional branches. In Spectre-PHT, first introduced as Spectre variant 1 by Kocher et al. [26], the attacker abuses the branch predictor to intentionally mispeculate 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. Store-to-Load (STL) dependencies require that loads do not execute before all stores have completed execution. 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 [30]. This behavior is exploited in the Spectre-STL [13] 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 [5] is a recent security property that extends constant-time to take transient executions into account.

Definition 1 (Speculative constant-time [5])**.** *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 [31]) and thus requires appropriate tools to efficiently model pairs of traces.

Binary-level symbolic execution. Symbolic Execution (SE) [17], [18], [32] 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 [33]. SE can also check assertions in order to *find bugs* or perform *bounded-verification* (i.e., verification up to a certain depth).

The common practice to analyze binary code is to decode instructions into an intermediate low-level language—here DBA [34] (see Appendix A). In binary-level SE, values (e.g. registers, memory addresses, memory content) are fixed-size symbolic bitvectors [35]. The memory is represented as a symbolic array of bytes addressed with 32-bit bitvectors. A symbolic array is a function $(Array\ \mathcal{I}\ \mathcal{V})$ mapping each index $i \in \mathcal{I}$ to a value $v \in \mathcal{V}$ with operations:

- $select : (Array\ \mathcal{I}\ \mathcal{V}) \times \mathcal{I} \rightarrow \mathcal{V}$ takes an array a and an index i and returns value v stored at index i in a,
- $store : (Array\ \mathcal{I}\ \mathcal{V}) \times \mathcal{I} \times \mathcal{V} \rightarrow (Array\ \mathcal{I}\ \mathcal{V})$ takes an array a, an index i, and a value v, and returns the array a in which i maps to v.

Relational Symbolic Execution (RelSE). RelSE [16], [36] is a promising approach to extend SE for analyzing security properties of two execution traces such as SCT[1]. 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, checks whether the outcomes of conditional branches and the memory indexes must be equal in both execution—meaning that they do not depend on the secret, or not.

[1]Tainting can also be used to approximate such properties but is less precise (see Section VII).

In RelSE, variables are mapped to *relational expressions* which are either *pairs* of symbolic expressions (denoted $\langle \varphi_l \,|\, \varphi_r \rangle$) when they *may* depend on secret input; or *simple* symbolic expressions (denoted $\langle \varphi \rangle$) when the *do not* depend on secret input. For the security evaluation of memory accesses and conditional instructions, we use the function *secLeak* defined in [16] which ensures that a relational expression does not depend on secrets (i.e. that its left and right components are necessarily equal):

$$\textit{secLeak}\,(\widehat{\varphi}, \pi) = \begin{cases} true & \text{if } \widehat{\varphi} = \langle \varphi \rangle \\ true & \text{if } \widehat{\varphi} = \langle \varphi_l \,|\, \varphi_r \rangle \wedge \not\models (\pi \wedge \varphi_l \neq \varphi_r) \\ false & \text{if } \widehat{\varphi} = \langle \varphi_l \,|\, \varphi_r \rangle \wedge \models (\pi \wedge \varphi_l \neq \varphi_r) \end{cases}$$

where $\models$ (resp. $\not\models$) denotes (un-)satisfiability. It relies on the fact that, if $\widehat{\varphi}$ is a simple expression then, by definition, it does not depend on the secret and can be leaked securely. However, if $\widehat{\varphi}$ is a pair of expressions, $\langle \varphi_l \,|\, \varphi_r \rangle$, the leak is secure if and only if φ_l and φ_r cannot be distinct under the current path predicate π (i.e. $\pi \wedge \varphi_l \neq \varphi_r$ is unsatisfiable).

Notations. The set of symbolic bitvectors of size n is denoted $\mathcal{B}v_n$. The set of symbolic formulas is denoted Φ and $\varphi, \varphi_l, \varphi_r, \psi, \ldots$ are symbolic expressions (bitvectors or arrays) in Φ. The set of relational formulas is denoted $\mathbf{\Phi}$ and $\widehat{\varphi}, \widehat{\psi} \ldots$ are relational expressions in Φ. We denote $\widehat{\varphi}_{|l}$ (resp. $\widehat{\varphi}_{|r}$), the projection on the left (resp. right) value of $\widehat{\varphi}$. If $\widehat{\varphi} = \langle \varphi \rangle$, $\widehat{\varphi}_{|l}$ and $\widehat{\varphi}_{|r}$ are both defined as φ. We also lift the functions $select$ and $store$ on symbolic arrays to relational expressions:

- $\mathbf{select}(\widehat{\mu}, \widehat{\imath}) \triangleq \langle select(\widehat{\mu}_{|l}, \widehat{\imath}_{|l}) \,|\, select(\widehat{\mu}_{|r}, \widehat{\imath}_{|r}) \rangle$
- $\mathbf{store}(\widehat{\mu}, \widehat{\imath}, \widehat{\nu}) \triangleq \langle store(\widehat{\mu}_{|l}, \widehat{\imath}_{|l}, \widehat{\nu}_{|l}) \,|\, store(\widehat{\mu}_{|r}, \widehat{\imath}_{|r}, \widehat{\nu}_{|r}) \rangle$.

III. Haunted RelSE

To analyze speculative constant-time, we need to modify RelSE to consider the speculative semantics of the program [5]. This includes *regular executions*—instructions that are executed as a result of a good speculation and are kept once the speculation is resolved—and all possible *transient executions*—instructions that are executed as a result of a mispeculation and that are discarded once the speculation is resolved. This section illustrates the straightforward approach to the problem—employed in state-of-the art tools (see Table V)—that we call *Explicit* as it models transient executions explicitly, and presents our optimized exploration strategy that we call *Haunted*.

A. Spectre-PHT

1) Explicit RelSE for Spectre-PHT: The *Explicit* approach to model Spectre-PHT in SE—introduced in KLEESpectre [11]—explicitly models transient executions by forking into four path at each conditional branch. Consider for instance, the program in Fig. 1a and its symbolic execution tree in Fig. 1b. After the conditional instruction **if** `c1` the execution forks into four paths:

- Two *regular paths*: Like in standard symbolic execution, the first path follows the $then$ branch and adds the constraint $(c_1 = true)$ to the path predicate; while the second path follows the $else$ branch with the constraint $(c_1 = false)$.
- Two *transient paths*: To account for transient executions that are mispredicted to $true$, the $then$ branch is executed with the constraint $(c_1 = false)$; while to account for transient executions that are mispredicted to $false$, the $else$ branch is executed with the constraint $(c_1 = true)$. These transient paths are discarded after reaching a *speculation bound* (usually defined by the size of the reorder buffer).

To verify speculative constant-time, we have to check that memory accesses and conditional statements do not leak secret information on both regular paths and transient paths. On regular paths, we check that the *control-flow* of the program and the *indexes of load and store* instructions do not depend on the secret input. However, on transient paths, we only check the *control-flow* and the *index of load* instructions. Reason is that, in speculative execution, memory stores are queued in the store buffer and are invisible to the cache until they are retired [11].

Problem with Explicit: From Fig. 1b, we see that both subtrees resulting from executing the $then$ branch in regular and transient execution (i.e. subtrees starting from state `A`) correspond to the same instructions under different path predicates. Precisely, if we call $\widehat{\psi}_{cf}$, $\widehat{\psi}_{ld}$, and $\widehat{\psi}_{st}$ the relational expressions corresponding respectively to control-flow statements, load indexes and store indexes in subtree `A`, then we have to check $secleak(\pi \wedge c_1, \widehat{\psi}_{cf}) \wedge secleak(\pi \wedge c_1, \widehat{\psi}_{ld}) \wedge secleak(\pi \wedge c_1, \widehat{\psi}_{st})$ for the regular execution, and $secleak(\pi \wedge \neg c_1, \widehat{\psi}_{cf}) \wedge secleak(\pi \wedge \neg c_1, \widehat{\psi}_{ld})$ for the transient execution. In the end, this is equivalent to checking the formula:

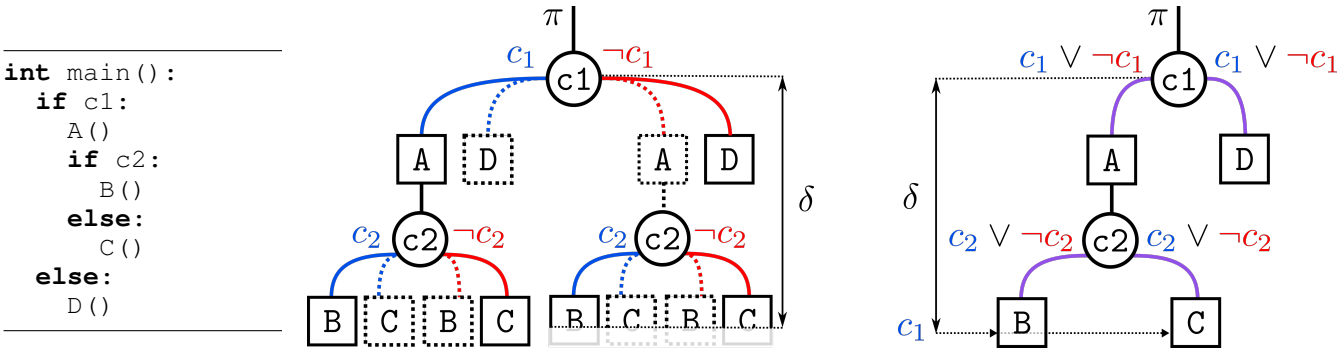
$$secleak(\pi, \widehat{\psi}_{cf}) \wedge secleak(\pi, \widehat{\psi}_{ld}) \wedge secleak(\pi \wedge c_1, \widehat{\psi}_{st})$$

This formula essentially amounts to symbolically executing the $then$ branch up to δ, checking load indexes $\widehat{\psi}_{ld}$ and control-flow expressions $\widehat{\psi}_{cf}$ without adding the constraint c_1, and only add c_1 to check store indexes $\widehat{\psi}_{st}$.

This observation led us to design an optimization of Explicit RelSE: we can explore a single speculative path *that encompasses both the* regular *and the* transient *behavior of the program in order to prune states while keeping an equivalent result.*

2) Haunted RelSE for Spectre-PHT: Instead of forking the execution into four paths, Haunted RelSE forks the execution into two paths, as illustrated in Fig. 1c. After conditional branch **if** `c1`, the execution forks into two paths: a path following the $then$ branch (subtree `A`) and a path following the $else$ branch (subtree D). Both paths model the behavior of the regular and the corresponding transient paths at the same time. Moreover, it delays (and possibly spares) satisfiability check of the path constraint—the constraint $c_1 \vee \neg c_1$ is added only for clarity. Finally, the constraint is added to the path predicate when the conditional branch is retired (after δ steps).

At each conditional statement (resp. load instruction), we check that the condition (resp. load index) does not depend on the secret in both the *regular and transient* executions (i.e. using path predicate π): $secleak(\pi, \widehat{\psi}_{cf}) \wedge secleak(\pi, \widehat{\psi}_{ld})$. On the other hand, store instructions are checked under the *regular*



(a) Illustrative program. (b) Explicit RelSE. (c) Haunted RelSE.

Figure 1: Comparison of RelSE of program in Fig. 1a, where solid paths represent regular executions, dotted paths represent transient executions, and δ is the speculation depth.

execution only (i.e. using path predicate $\pi \wedge c_1$): $secleak(\pi \wedge c_1, \widehat{\psi}_{st})$. Finally, the condition $(c_1 = true)$ is added to the path predicate after δ steps.

B. Spectre-STL

1) Explicit RelSE for Spectre-STL: At the microarchitectural level, a load instruction can take its value from any matching entry in the store buffer, or from the main memory. In other words, the load can bypass each pending store in the store buffer until it reaches the main memory. To account for this behavior, the *Explicit* strategy—employed in PITCHFORK [5]—is to models transient executions explicitly by forking the symbolic execution for each possible load and store interleaving.

Consider as an illustration the program in Fig. 2a. Symbolic execution of the store intructions gives the symbolic memory μ_3 defined in Fig. 2b which is the sequence of symbolic store operations starting from $initial\_memory$. With this chronological representation, we can easily define the content of a *store buffer* of size $|SB|$ by taking the $|SB|$ last store operations of the symbolic memory. Similarly, the *main memory* can be defined by removing the last $|SB|$ store operations from the symbolic memory. If we consider a store buffer of size 2, the last two store expressions constitute the store buffer while the main memory is defined by μ_1.

The first load instruction (block A) can bypass each store operation in the store buffer until it reaches the main memory. Therefore there are three possible values for x, as detailed in Fig. 2c:

- The *regular value* r corresponds to a symbolic $select$ operation from the most recent symbolic memory μ_3. Because all prior store operations are encoded in-order into μ_3, this corresponds to the in-order execution.
- The first *transient value* t_2 is obtained by bypassing the first entry in the store buffer. This corresponds to a symbolic $select$ operation from μ_2.
- The final *transient value* t_1 is obtained by bypassing the first and the second entries in the store buffer and taking the value from the main memory. This corresponds to a symbolic $select$ operation from μ_1.

Similarly, variable y can also take three possible values.

The *Explicit* exploration strategy, illustrated in Fig. 2d, forks the symbolic execution for each possible value that a load can take. This quickly leads to path explosion and we show experimentally (Section V-C) that this solution is intractable even on small codes (100 instr.).

2) Haunted RelSE for Spectre-STL: The first observation that we make is that most paths are redundant as a load can naturally commute with non-aliasing prior stores. Take, for instance, the evaluation of loads in Fig. 2c. If we can determine that the index a of the load is distinct from the index of the second store a_2 then, by the theory of arrays, we have $t_2 = t_1$ thus the path $x \mapsto t_2$ and all of its subpaths are redundant. *We rely on a well-known optimization for symbolic arrays called read-over-write [37] to detect and prune these redundant cases.*

However, merely pruning redundant cases is not sufficient to deal with path explosion (see Section V), thus *we propose a new encoding to keep the remaining cases in a single path predicate*. We use symbolic *if-then-else* to encode in a single expression all the possible values that a load can take instead of forking the execution for each possible case.

Take, for instance, the evaluation of load expressions given in Fig. 2c. After the evaluation of the second load, the variable y can take the values r', t'_1, or t'_2. We introduce two fresh boolean variables b'_1 and b'_2 and build the expression $(ite\ b'_1\ t'_1\ (ite\ b'_2\ t'_2\ r'))$. The solver can let y take the following values:

- transient value t'_1 by setting b'_1 to true,
- transient value t'_2 by setting b'_1 to false and b'_2 to true,
- regular value r by setting both b'_1 and b'_2 to $false$.

Finally, transient values t'_1 (resp. t'_2) can easily be discarded (e.g. after reaching the speculation depth) by setting b'_1 (resp. b'_2) to false.

IV. Implementation of Haunted RelSE

This section introduces the technical details of Haunted RelSE. It mainly focuses on the changes to binary-level RelSE [16] required to analyze speculative constant-time [5].

```
store a1 v1;   } S
store a2 v2;
store a3 v3;
x = load a;    } A
y = load b;    } B
[...]          } C
```

(a) Illustrative program.

$$\left.\begin{array}{l}\mu_0 = initial\_memory\\ \mu_1 = store\ \mu_0\ a_1\ v_1\end{array}\right\}\text{Mem}$$
$$\left.\begin{array}{l}\mu_2 = store\ \mu_1\ a_2\ v_2\\ \mu_3 = store\ \mu_2\ a_3\ v_3\end{array}\right\}\text{SB}$$

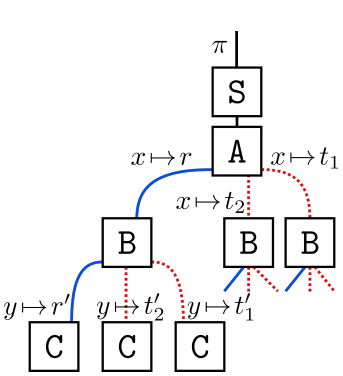
(b) Symbolic memory where SB is the store buffer (of size 2) and Mem is the main memory.

$$x = \left\{\begin{array}{ll} r & \text{In-order execution}\\ t_2 & \text{Bypass 1}^{\text{st}}\text{ SB entry}\\ t_1 & \text{Bypass 1}^{\text{st}}\text{ \& 2}^{\text{nd}}\text{ SB entries}\end{array}\right.$$
$$y = \{r', t'_1, t'_2\}$$

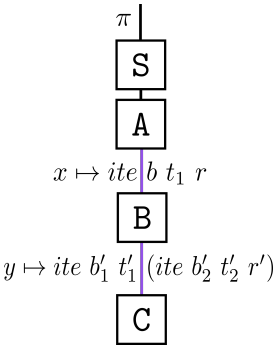
where

$$r = select\ \mu_3\ a, \qquad r' = select\ \mu_3\ a'$$
$$t_2 = select\ \mu_2\ a, \qquad t'_2 = select\ \mu_2\ a'$$
$$t_1 = select\ \mu_1\ a, \qquad t'_1 = select\ \mu_1\ a'$$

(c) Symbolic evaluation of loads.



(d) Explicit RelSE.



(e) Haunted RelSE when $a \neq a_2$.

Figure 2: Speculative RelSE of program in Fig. 2a. The symbolic memory is given in Fig. 2b and the symbolic evaluation of load instructions is detailed in Fig. 2c. Figure 2d illustrates the symbolic execution tree obtained from the Explicit exploration strategy; and Fig. 2e, the tree obtained from Haunted RelSE, where solid paths denote regular executions and dotted paths denote transient executions.

Most instructions naturally commute or cannot be reordered because of their data dependencies. Indeed, we only need to consider reordering of conditional branches for Spectre-PHT (Section IV-A) and reordering of load and store instructions for Spectre-STL (Section IV-B).

A symbolic configuration, denoted σ, consists of:

- the current location l, which is used to get the current instruction in the program P, denoted $P[l]$;
- the current depth of the symbolic execution δ;
- a symbolic register map ρ, mapping program variables to their symbolic value;
- two path predicates π and $\widetilde{\pi}$ (details in Section IV-A);
- a symbolic memory $\widehat{\mu}$—a pair of symbolic arrays and the *retirement depth* of its store operations;
- a set of transient loads $\widetilde{\lambda}$, (details in Section IV-B).

The notation $\sigma.f$ is used to denote the field f in configuration σ. We also define a function *eval_expr*(σ, e) which evaluates a DBA expression e to a symbolic value in a symbolic configuration σ.

Instead of modeling the reorder buffer explicitly, we use the *current depth* of the symbolic execution to track instructions to retire. An instruction must be retired after at most Δ steps, where Δ is the size of the reorder buffer. Expressions are annotated with a depth to determine when they must be retired, or whether they depend on the memory. For instance, a variable v in the register map ρ, maps to a pair $(\widehat{\varphi}, \delta)$ where δ is the retirement depth of its last memory access. When δ is not needed in the context, it is omitted.

A. Haunted RelSE for Spectre-PHT.

1) Evaluation of conditional instructions: Contrary to standard symbolic execution, conditions are not added to the path predicate right away. Instead, they are kept in a *speculative path predicate*, denoted $\widetilde{\pi}$, along with their retirement depth. When the retirement depth of a condition is reached, it is removed from the speculative path predicate and added to the *retired path predicate*, denoted π.

Evaluation of conditional branches is detailed in Algorithm 1. First, the function evaluates the symbolic value of the condition and checks that it can be leaked securely. Then it computes the two next states σ_t, following the $then$ branch, and σ_f, following the $else$ branch by updating the location and the speculative path predicate $\widetilde{\pi}$.

Func *eval_ite*(σ) **where** $P[\sigma.l] =$ `ite c ? l_t : l_f` **is**
  $(\widehat{\varphi}, \delta) \leftarrow$ *eval_expr*(σ, c);
  assert *secLeak* $(\widehat{\varphi}, \sigma.\pi)$; ▷ *Leakage of c is secure*
  ▷ *Compute state following then branch*
  $\sigma_t \leftarrow \sigma$; $\sigma_t.l \leftarrow l_t$; $\sigma_t.\widetilde{\pi} \leftarrow \sigma.\widetilde{\pi} \cup \{(\widehat{\varphi}, \delta)\}$;
  ▷ *Compute state following else branch*
  $\sigma_f \leftarrow \sigma$; $\sigma_f.l \leftarrow l_f$; $\sigma_f.\widetilde{\pi} \leftarrow \sigma.\widetilde{\pi} \cup \{(\neg\widehat{\varphi}, \delta)\}$;
  return (σ_t, σ_f)

Algorithm 1: Evaluation of conditional branches.

2) Determining speculation depth: The speculation depth after a conditional branch is computed dynamically, considering that the condition can be fully resolved (and mispredicted paths can be squashed) when all the memory accesses upon which it depends are retired. In particular it means that if the condition does not depend on the memory then the branch is not mispredicted [9], [10].

This requires to keep, for each expression, the depth of its last memory access. As shown in Algorithm 1, at a conditional branch `ite c ? l_true : l_false`, `c` evaluates to a symbolic value $\widehat{\varphi}$ and depth δ. This depth δ is added to the speculative path predicate $\widetilde{\pi}$ as the retirement depth of the condition.

3) Invalidate transient paths: Conditional branches are retired in the function *retire*PHT$(\pi, \widetilde{\pi}, \delta)$ defined in Algorithm 2. The function removes from the speculative path predicate $\widetilde{\pi}$ all the conditions with a retirement depth δ_{ret} below the current depth $\delta_{current}$, and adds them to the retired path predicate π.

It returns the updated path predicates π and $\widetilde{\pi}$. The symbolic execution stops, if π becomes unsatisfiable.

Func *retire*$_{\text{PHT}}(\pi, \widetilde{\pi}, \delta_{current})$ **is**
  $\pi' \leftarrow \pi$; $\widetilde{\pi}' \leftarrow \emptyset$;
  for $(\widehat{\varphi}, \delta_{ret})$ **in** $\widetilde{\pi}$ **do**
    if $\delta_{ret} \leq \delta_{current}$ **then** ▷ *Condition to retire*
      $\pi' \leftarrow \pi' \wedge \widehat{\varphi}$;
    else $\widetilde{\pi}' \leftarrow \widetilde{\pi}' \cup \{(\widehat{\varphi}, \delta_{ret})\}$;
  return $(\pi', \widetilde{\pi}')$

Algorithm 2: Retire expired conditions.

B. Haunted RelSE for Spectre-STL.

1) Symbolic memory: In a symbolic configuration, the memory $\widehat{\mu}$ is the history of symbolic store operations starting from the initial memory. We can use this chronological representation to reconstruct the content of the *store buffer* and the *main memory*. The *store buffer*, is the restriction of the symbolic memory to the last $|SB|$ stores which have not been retired, where $|SB|$ is the size of the store buffer. Formally, it is defined as:

$$SB(\widehat{\mu}, \delta) \triangleq \{(s, \delta_s) \mid (s, \delta_s) \in \mathit{last}(|SB|, \widehat{\mu}) \wedge \delta > \delta\}$$

where $\mathit{last}(n, \widehat{\mu})$ is the projection on the last n element of the symbolic memory $\widehat{\mu}$.

Similarly, the *main memory* is defined as the restriction of the symbolic memory to the retired store operations. Formally, it is defined as $Mem(\widehat{\mu}, \delta) \triangleq \widehat{\mu} \setminus SB(\widehat{\mu}, \delta)$

The evaluation of a store instruction is detailed in Algorithm 3. First, the function evaluates the symbolic values of the index and check that it can be leaked securely under the *regular path predicate* π_{reg} (i.e. the conjunction of the retired path predicate π with all the pending conditions in $\widetilde{\pi}$, plus the invalidation of the transient loads in $\widetilde{\lambda}$). Then, it updates the symbolic memory with a symbolic $store$ operation and sets the retirement depth of this store to $\delta + \Delta$. This retirement depth is used to determine which store operations are pending in the store buffer and which ones are committed to the main memory.

Func *eval_store*(σ) **where** $P[\sigma.l] = \texttt{store}\ i\ v$ **is**
  $\hat{\imath} \leftarrow$ *eval_expr*(σ, i);
  $\widehat{\nu} \leftarrow$ *eval_expr*(σ, v);
  $\pi_{reg} \triangleq$ *retire*$_{\text{ALL}}(\sigma.\pi, \sigma.\widetilde{\pi}, \sigma.\widetilde{\lambda}, \sigma.\delta)$;
  assert *secLeak*$(\hat{\imath}, \pi_{reg})$; ▷ *Leakage of i is secure*
  $\sigma' \leftarrow \sigma$; $\sigma'.l \leftarrow \sigma.l + 1$;
  ▷ *Update memory, store will retire after* Δ *steps*
  $\sigma'.\widehat{\mu} \leftarrow (\mathbf{store}(\sigma.\widehat{\mu}, \hat{\imath}, \widehat{\nu}), \delta + \Delta)$;
  return σ'

Algorithm 3: Evaluation of store instructions where *retire*$_{\text{ALL}}$ returns the regular path predicate (details in Section IV-B3).

2) Evaluation of load expressions: Load expressions can either take their value from a pending store in the store buffer with a matching address via *store-to-load forwarding*; or can speculatively bypass pending stores in the store buffer and take their value from the main memory [30]. Instead of considering all possible interleavings between a load expression and prior stores in the store-buffer, we use *read-over-write* [37] to identify and discard most cases in which the load and a prior store naturally commute. *Read-over-write* is a well known simplification for the theory of arrays which resolves select operations on symbolic arrays ahead of the solver.

To efficiently compare indexes, read-over-write relies on *syntactic term equality*. The comparison function $\mathit{eq}^{\#}(i, j)$ returns true (resp. false) only if i and j are syntactically equal (resp. different). If the terms are not comparable, it is undefined, denoted $\perp$.

To efficiently resolve select operations ahead of the solver, read-over-write defines a lookup_{mem} function relying on this syntactic term equality:

$$\mathit{lookup}_{mem}(\widehat{\mu}_0, i) \triangleq \mathbf{select}(\widehat{\mu}_0, i)$$
$$\mathit{lookup}_{mem}(\widehat{\mu}_n, i) \triangleq \begin{cases} \widehat{\varphi} & \text{if } \mathit{eq}^{\#}(i, j) \\ \mathit{lookup}_{mem}(\widehat{\mu}_{n-1}, i) & \text{if } \neg\mathit{eq}^{\#}(i, j) \\ \mathbf{select}(\widehat{\mu}_n, i) & \text{if } \mathit{eq}^{\#}(i, j) = \perp \end{cases}$$

where $\widehat{\mu}_n \triangleq \mathbf{store}(\widehat{\mu}_{n-1}, j, \widehat{\varphi})$.

Example: Consider a memory $\widehat{\mu}$ such that:

$$\widehat{\mu} \quad = \quad \boxed{ebp - 8 \mid \widehat{\varphi}} \longrightarrow \boxed{ebp - 4 \mid \widehat{\psi}} \longrightarrow \boxed{eax \mid \widehat{\phi}} \longrightarrow [\,]$$

- $\mathit{lookup}_{mem}(\widehat{\mu}, ebp - 8)$ returns $\widehat{\varphi}$.
- $\mathit{lookup}_{mem}(\widehat{\mu}, ebp - 4)$ first compares indexes $ebp - 4$ and $ebp - 8$ and determines that they are *syntactically distinct* ($\neg\mathit{eq}^{\#}(ebp - 4, ebp - 8)$). It then moves to the second element, returns $\widehat{\psi}$.
- $\mathit{lookup}_{mem}(\widehat{\mu}, eax)$ compares indexes $ebp - 8$ and eax but, without further information, the equality or disequality of ebp and eax cannot be determined ($\mathit{eq}^{\#}(ebp - 4, eax) = \perp$). Therefore the $select$ operation cannot be simplified.

In order to model store-to-load forwarding efficiently, we define an new function lookup_{SB} in Algorithm 4 which returns a set of value from matching stores in the store buffer. Additionally, lookup_{SB} returns the depth at which each load must be invalidated, that is, the retirement depth of a most recent store to the same address.

Func $\mathit{lookup}_{SB}(SB, Mem, i)$ **is**
  $S \leftarrow \emptyset$; $\delta \leftarrow \infty$;
  ▷ *Load from store buffer*
  for $((\mathbf{store}(\widehat{\mu}, j, \widehat{\varphi}), \delta')$ **in** $SB)$ **do**
    if $\mathit{eq}^{\#}(i, j) = true$ **then** ▷ *Must alias*
      $S \leftarrow S \cup \{(\widehat{\varphi}, \delta)\}$; $\delta \leftarrow \delta'$;
    else if $\mathit{eq}^{\#}(i, j) = \perp$ **then** ▷ *May alias*
      $S \leftarrow S \cup \{(\mathbf{select}(\widehat{\mu}, i), \delta)\}$; $\delta \leftarrow \delta'$;
    else continue; ▷ *Must not alias*
  ▷ *Load from main memory*
  $S \leftarrow S \cup \{(\mathit{lookup}_{mem}(Mem), \delta)\}$;
  return S

Algorithm 4: Definition of lookup_{SB}

Finally, we define a function $\mathit{lookup}_{ite}(\widehat{\mu}, i, \widetilde{\lambda}, \delta)$ in Algorithm 5 which encodes the result of lookup_{SB} as a symbolic if-then-else expression using fresh boolean variables. The function returns the value of the load expression, and adds the boolean variables declared in the process to $\widetilde{\lambda}$. Additionally, in the implementation of BINSEC/HAUNTED, we use the name of the boolean variables to encode information about the location of the load and of the forwarding store. Therefore, using the counterexample returned by the solver it is possible to understand which stores have been bypassed to trigger the violation. This helps users understand the violation and reconstruct the attack graph [38].

Func $\mathit{lookup}_{ite}(\widetilde{\lambda}, \widehat{\mu}, i, \delta)$ **is**
  $S \leftarrow \mathit{lookup}_{SB}(SB(\widehat{\mu}, \delta), Mem(\widehat{\mu}, \delta), i)$;
  $S, \widehat{\nu} \leftarrow get_{\infty}(S)$; ▷ *Get regular value from S*
  for $(\widehat{\varphi}, \delta_{ret})$ **in** S **do**
    $b \leftarrow fresh\_boolean\_var$;
    $\widehat{\nu} \leftarrow ite\ \beta\ then\ \widehat{\varphi}\ else\ \widehat{\nu}$;
    $\widetilde{\lambda} \leftarrow \widetilde{\lambda} \cup \{(\beta, \delta_{ret})\}$; ▷ *Save retire depth*
  return $\nu, \widetilde{\lambda}$

Algorithm 5: Definition of lookup_{ite}

The evaluation of a load instruction is detailed in Algorithm 6. First, the function evaluates the symbolic values of the index and check that it can be leaked securely. Then it calls lookup_{ite}, which returns the set of symbolic values that the load can take, encoded as a single if-then-else expression $\hat{\iota}$ and updates the set of transient load $\widetilde{\lambda}$. Finally, it updates the register map with the load value and sets its retirement depth to $\delta + \Delta$. Retirement depth is later used in the evaluation of conditional branches to determine whether the condition might depend on memory.

Func *eval_load*(σ) **where** $P[\sigma.l] = v$ `:= load` i **is**
  $\hat{\iota} \leftarrow \mathit{eval\_expr}(\sigma, i)$;
  assert $\mathit{secLeak}(\hat{\iota}, \sigma.\pi)$; ▷ *Leakage of i is secure*
  $\widehat{\nu}, \widetilde{\lambda}' \leftarrow \mathit{lookup}_{ite}(\sigma.\widetilde{\lambda}, \sigma.\widehat{\mu}, \hat{\iota}, \sigma.\delta)$;
  $\sigma' \leftarrow \sigma$; $\sigma'.l \leftarrow \sigma.l + 1$; $\sigma'.\widetilde{\lambda} \leftarrow \widetilde{\lambda}'$;
  ▷ *Update v with load value and current depth*
  $\sigma'.\rho \leftarrow \sigma.\rho[\mathrm{v} \mapsto (\widehat{\nu}, \sigma.\delta + \Delta)]$;
  return σ'

Algorithm 6: Evaluation of load instructions.

3) Invalidate transient loads: Transient load values can be invalidated when more recent matching stores are retired by setting the corresponding boolean variables to $false$. The function $\mathit{retire}_{\mathrm{STL}}(\pi, \widetilde{\lambda}, \delta)$, defined in Algorithm 7, removes from the set of transient loads $\widetilde{\lambda}$ all the loads with an invalidation depth below δ, and set the corresponding booleans to $false$ in the path predicate π. For readability, we introduce a function $\mathit{retire}_{\mathrm{ALL}}$ which stops all speculation by applying both $\mathit{retire}_{\mathrm{PHT}}$ and $\mathit{retire}_{\mathrm{STL}}$.

C. Theorems

In this section we prove that Haunted RelSE is correct and complete (up-to-an-unrolling-bound) for SCT. This means

Func $\mathit{retire}_{\mathrm{STL}}(\pi, \widetilde{\lambda}, \delta_{current})$ **is**
  $\pi' \leftarrow \pi$; $\widetilde{\lambda}' \leftarrow \emptyset$;
  for (β, δ_{ret}) **in** $\widetilde{\lambda}$ **do**
    if $\delta_{ret} \leq \delta_{current}$ **then** ▷ *Load to retire*
      $\pi' \leftarrow \pi' \wedge (\beta = false)$;
    else $\widetilde{\lambda}' \leftarrow \widetilde{\lambda}' \cup \{(\beta, \delta_{ret})\}$;
  return $(\pi', \widetilde{\lambda}')$

Algorithm 7: Retire expired load values.

that when Haunted RelSE reports a violation, it is a real violation of SCT (no over-approximation); and when it reports no violations up to depth k then the program is secure up to depth k (no under-approximation). To this end, we prove that Haunted RelSE is equicvalent to Explicit RelSE and show that Explicit RelSE is correct and complete up-to-an-unrolling-bound for SCT.

Theorem 1. *Explicit RelSE is correct and complete up-to-an-unrolling-bound for speculative constant-time.*

Proof (Sketch). The proof is a simple extension of the proofs of correctness and completeness of RelSE for constant-time [16], to the speculative semantics. The extension requires to show that 1) violations reported on transient paths in the symbolic execution correspond to violations in concrete transient execution, and 2) if there is a violation in concrete transient execution, then there is a path in symbolic execution that reports this violation. □

Next, we show that Haunted RelSE is equivalent to Explicit RelSE.

Theorem 2 (Equivalence Explicit and Haunted RelSE)**.** *Haunted RelSE detects a violation in a program if and only if Explicit RelSE detects a violation.*

A sketch a proof is given in Appendix B. We first show that the theorem holds for Spectre-PHT: after a conditional branch, the two paths explored in Haunted RelSE fully capture the behavior of the four paths explored in Explicit RelSE. Then we show that it holds for Spectre-STL: after a load instruction, the single path resulting from Haunted RelSE fully captures the behavior of the multiple paths explored in Explicit RelSE.

Corollary 1. *Haunted RelSE is correct and complete up-to-an-unrolling-bound for speculative constant-time.*

D. BINSEC/HAUNTED, a tool for Haunted RelSE

We implement Haunted RelSE on top of the binary-level analyzer BINSEC [39] in a tool named BINSEC/HAUNTED[2]. BINSEC/HAUNTED takes as input an x86 executable, the location of secret inputs, an initial memory configuration (possibly fully symbolic), the speculation depth, and size of the store buffer. BINSEC/HAUNTED explores the program in a depth-first search manner, prioritizing transient paths over regular paths, and reports SCT violations with counterexamples (i.e., initial configurations and speculation choices leading to the violation). It uses the SMT solver Boolector [40], currently the best for the theory of bitvectors [37], [41].

[2]Open sourced at: https://github.com/binsec/haunted

V. Experimental Evaluation

We answer the following research questions:

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

RQ2 **Haunted vs. Explicit.** How does *Haunted RelSE* compares against *Explicit RelSE*?

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

To answer RQ1 and RQ2, we compare the performance of *Explicit* and *Haunted* explorations strategies for RelSE—both implemented in BINSEC/HAUNTED — on a set of real word cryptographic binaries and litmus benchmark (Sections V-B and V-C). To answer RQ3, we compare BINSEC/HAUNTED against state-of-the-art competitors, KLEESpectre [11] and Pitchfork [5] (Section V-D).

Legend. In this section, I_{x86} is the number of *unique* x86 instructions explored, P is the number of paths explored, T is the overall execution time, 🐞 is the number of violations (i.e. the number instructions leaking secret data), ⧗ is the number of timeouts, ✓ is the number of programs proven secure, ✗ is the number of programs proven insecure.

A. Benchmarks

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 [5]), 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 *in-order* execution and implement a *shadow stack* to constrain return instructions to their proper return site. Considering transient jump targets requires to model indirect jumps on arbitrary locations, which is doable but intractable for symbolic execution.

We evaluate BINSEC/HAUNTED on the following programs:

- `litmus-pht`: 16 small test cases (litmus tests) for Spectre-PHT taken from Pitchfork's modified set of Paul Kocher's litmus tests[3],
- `litmus-pht-patched`: `litmus-pht` that we patched with index masking [19],
- `litmus-stl`: our new set of litmus tests for Spectre-STL,[4]
- Cryptographic primitives from OpenSSL and Libsodium cryptographic libraries (detailed in Table I), including and extending those analyzed in [5].

Programs are compiled statically for a 32-bit x86 architecture with `gcc 10.1.0`. Litmus tests are compiled with options `-fno-stack-protector` and Spectre-STL litmus tests are additionally compiled with `-no-pie` and `-fno-pic` in order to rule out violations introduced by these options (see Section VI). For the same reason, `donna` and `tea` are compiled without stack protectors `-fno-stack-protector` and for optimization levels `O0`, `O1`, `O2`, `O3`, and `Ofast`. Libsodium is compiled with the default Makefile and OpenSSL is compiled with optimization level `O3` (both including stack protector).

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.

Note on Stack Protectors: Error-handling code introduced by stack protectors is complex and contains many syscalls 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).

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

We compare the performance of Haunted RelSE and Explicit RelSE—that we call 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. To show the importance of Haunted RelSE for path pruning in programs containing loops, we also detail the execution of a litmus test containing a loop (`case_5`) in Appendix C.

Results. For `litmus-pht` and `litmus-pht-masked`, we can see that Haunted RelSE:

- explores less paths ($4\times$) for an equivalent result, limiting path explosion (see Appendix C),
- analyzes programs faster ($1437\times$ and $21\times$ respectively), achieving performance in line with NoSpec,
- can fully explore 2 additional programs and finds 1 more violation whereas Explicit RelSE times-out.

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

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

[4]Open sourced at: https://github.com/binsec/haunted_bench

[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 [42]

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	911	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: Experiments for Spectre-PHT with Spectre-STL disabled and speculation bound computed dynamically.

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	-
	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	**200**	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	-
	Explicit	**2k**	93M	108004	**22**	**15**	**3/4**	**13/23**
	Haunted	**17k**	42	88078	**148**	**8**	**4/4**	**23/23**

Table III: Experiments for Spectre-STL with support for Spectre-PHT disabled.

two other paths being unsatisfiable) which makes it equivalent to Haunted RelSE.

Finally, for Libsodium and OpenSSL, Explicit RelSE gets stuck exploring complex code introduced by stack protectors and spends most of its time checking satisfiability of the path constraint before timing out. Haunted RelSE circumvents this issue by delaying the update of the path constraint, thus it can *fully explore* `secretbox` and `ssl3-digest` without timing-out, with a noticeable speedup ($8.9\times$ and $4.6\times$), covering more code ($4.6\times$ and $3\times$), and finding 4 more violations. While Haunted RelSE times out on the more complex primitive `mee-cbc`, it still explores $3.5\times$ more code than Explicit.

Conclusion. While the Explicit strategy already allows to find Spectre-PHT violations in realistic codes, Haunted RelSE strongly improves the performance in terms of speed ($2.3\times$ faster in total, up to $1437\times$), timeouts (-66%) and covered code ($1.28\times$ in total, up to $4.6\times$). Actually, we can see that Haunted RelSE does not improve performance over Explicit RelSE in 3/7 benchmark families, but make a noticeable difference on the other 4/7 benchmark families (`litmus-pht`, `litmus-pht masked`, `secretbox`, `ssl3-digest`), where the performance gains become significant (from $4.6\times$ faster to $1437\times$).

C. Performance for Spectre-STL (RQ1-RQ2)

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

Results. The explosion of the number of paths for Explicit RelSE shows that the number of behaviors to consider for Spectre-STL grows exponentially. The performance of Explicit RelSE on litmus tests shows that encoding transient paths explicitly is not tractable—even though our implementation discards redundant paths. Overall, Haunted RelSE:

- scales better on `litmus-stl` tests and `tea`, achieving better analysis time (speed up of $3152\times$ and $3.4\times$), 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 ($8.6\times$ more unique instructions in total), finds 126 more violations and reports 10 more insecure programs.

Conclusion. While the state-of-the-art Explicit strategy shows low performance for Spectre-STL even on small programs (15 timeouts in total), Haunted RelSE strongly improves the performance in terms of speed ($1.2\times$ faster in total, up to $3152\times$), timeouts (8 vs. 15), covered code ($8.6\times$), number of violation found (+126) and number of programs deemed insecure (+10). Especially, 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).

D. Comparison with Pitchfork and KLEESpectre (RQ3)

KLEESpectre [11] is an adaptation of SE for finding Spectre-PHT violations[5], following an *Explicit exploration strategy*. It analyses LLVM bytecode while Pitchfork and BINSEC/HAUNTED analyze binary code, which gives KLEESpectre a *performance advantage*. Note that 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.

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

Pitchfork [5] is the only competing tool which can analyze programs for Spectre-STL. It is based on SE and *tainting* which is *faster than RelSE but also less precise* and can report false alarms (see Section VII). Pitchfork stops a path after finding a violation, while BINSEC/HAUNTED continues the execution. To provide a fair comparison, we also consider a modified version of Pitchfork, namely Pitchfork-cont, which does not stop after finding a violation.

Table IV reports the performance of KLEESpectre, Pitchfork, Pitchfork-cont and BINSEC/HAUNTED on `litmus-pht`, `litmus-pht-masked`, `tea`, and `donna`. We exclude `secretbox`, `ssl3-digest` and `mee-cbc` as the performance of the tools on these programs will vary according to how they handle syscalls. We report unique violations for each tool. We also exclude 6 spurious violations found by Pitchfork in non executable `.data` section after following a transient indirect jump.

	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.

We confirm, as reported by Cauligi *et al.* [5], that stack protectors introduce Spectre-PHT violations, and that `ret` instructions can be exploited with Spectre-STL to enable ROP-like attacks [27].

Result. KLEESpectre, as expected, shows similar results 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×). Also, it *fails to report 2 insecure litmus tests*: `case_7` and `case_10`. Program `case_10` contains an indirect leak while KLEESpectre only searches for direct leaks. Still, `case_7` contains a leak secret (LS) violation that KLEESpectre should report.

For Spectre-PHT, Pitchforks 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 perfomance and more timeouts. The analysis even 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 our tool can exhaustively explore small-size binaries, using realistic speculation windows. 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.

VI. NEW VULNERABILITIES AND MITIGATIONS

In this section, we report on: (a) potential problems with index-masking, a well-known defense against Spectre-PHT, and propose correct implementations to avoid them; and (b) new potential vulnerabilities introduced by popular gcc options. Programs are compiled with `gcc-10.2.0 -m32 -march=i386 -O0`. All vulnerabilities were automatically found by BINSEC/HAUNTED.

a) Index-masking defense: Index-masking [19] is a well known defense against Spectre-PHT—used in WebKit for example—which consists in strengthening conditional array bound checks with branchless bound checks. Indexes are masked with the length of the array, rounded up to the next power of two minus one. We give an example of index masking in Listing 3. For the array `publicarray` of size 16 the value of the mask is 15 (`0x0f`). For an arbitrary index `idx`, the masked index `(idx & 0x0f)` is strictly smaller than 16, hence the access is in bounds. This countermeasure prevents out-of-bound reads if the length of the array is a power of two and limits the scope of out-of-bound reads otherwise.

```
1 void leakThis(uint8_t toLeak) {
2   tmp &= publicarray2[toLeak * 512];
3 }
4 void case_1_masked(uint32_t idx) {
5   idx = idx & (publicarray_size - 1);
6   uint8_t toLeak = publicarray[idx];
7   leakThis(toLeak);
8 }
```

Listing 3: Illustration of index-masking

Using BINSEC/HAUNTED, we discover that whereas this countermeasure does protect against Spectre-PHT, it may also introduce new Spectre-STL vulnerabilities. Take for instance the compiled version of Listing 3, given in Listing 4. Line 1 computes the value of the mask and store it into `eax`. Line 2 performs the index masking and stores the masked index in the memory at `[ebp+idx]`. Line 3 loads the masked index into `eax`. *Notice that this load can bypass the store at line 2 and load the old unmasked index* `idx`. Then, line 4 loads the value at `publicarray[idx]` into `al`, allowing the attacker to read arbitrary memory—including secret data. Finally, the value of `al` is used as a load index at line 5, encoding

```
1  mov eax, publicarray_size - 1 ; Compute mask
2  and [ebp + idx], eax ; Store masked index
3  mov eax, [ebp + idx] ; Bypass prior store
4  mov  al, [@publicarray + eax] ; Out-of-bound load
5  mov  dl, publicarray2[al << 9] ; Leak secret
```

Listing 4: Compiled version of Listing 3 with `gcc-10.2.0 -m32 -march=i386 -O0`

secret data in the cache. To conclude, because the masked index is stored in the memory, the masking operation can be bypassed with Spectre-STL, leading to arbitrary memory read, and eventually leaking secret data.

This violation of SCT occurs at optimization level `O0` with both `clang-11.0` and `gcc-10.2` because the masked index is stored on the stack. We propose a *patched implementation* in Listing 5 that forces the index into a register (line 2) so the masking cannot be bypassed. A second solution is to set the optimization level to `O1` or higher so the store operation is optimized away—but this solution is fragile as it completely relies on compiler choices. In these two case, BINSEC/HAUNTED reports that the program is secure w.r.t. speculative constant-time.

```
1  void case_1_masked_patched(uint32_t idx) {
2    register uint32_t ridx asm ("edx");
3    ridx = idx & (publicarray_size - 1);
4    uint8_t toLeak = publicarray[ridx];
5    leakThis(toLeak); }}
```

Listing 5: Patch of index-masking for Spectre-STL

b) Position-independent-code: Position-independent code (PIC), and position-independent executables (PIE) are compiler options which makes it possible to load a binary to any memory location without modifying the code. These options are used to enable address space layout randomization (ASLR), which loads executables to non-predictable addresses in order to prevent a attackers from guessing target addresses, making return oriented programming (ROP) attacks more challenging. Our version of `gcc-10` compiles by default to position independent executables, which can be disabled by adding the options `-fno-pic -no-pie`.

Using BINSEC/HAUNTED, we have discovered that the code introduced by `gcc` in position independent executables *may introduce Spectre-STL vulnerabilities*. Indeed, on our set of STL-litmus-tests compiled with `-no-pie -fno-pic`, BINSEC/HAUNTED finds 13 violations and reports 4 programs as secure and 10 as insecure; whereas on STL-litmus-tests compiled without these options, it *finds 26 violations and reports only one program as secure*.

In x86, position independent executables access global variables as an offset from a *global pointer* which is set up at the beginning of the function, relatively to the current location. The current location is not directly accessible but is obtained via a function `x86_get_pc_thunk_ax` which loads its return address to `eax`. More precisely, a call to `x86_get_pc_thunk_ax` stores the return address on the stack before jumping to the function, then in the function this return address is loaded into `eax`. *With Spectre-STL, this load can bypass the previous store and load a stale value into* `eax`. Because `eax` is later used as a global offset, controlling its value, gives an attacker the ability to speculatively read at an arbitrary address. Take as an example the program in Listing 6, that we explain line per line:

```
1  __x86_get_pc_thunk_ax:
2    mov eax, [esp+0]; bypass stored @ret and
3    retn             ; load attacker controlled
4                     ; value 0x023f35
5  case_1_masked_patched:
6    call __x86_get_pc_thunk_ax; eax = 0x023f35
7    add  eax, 0x9E0FA          ; eax = 0x0c202f
8    mov  edx, (publicarray_size - 0x0A2000)[eax]
9    ; edx = [0x0C20EF] = secret
10   [...]
11   mov  dl, (publicarray - 0x0A2000h)[eax + edx]
12   ; Violation: secret dependent load
```

Listing 6: Compiled version of Listing 5, with PIC enabled. Secret data is stored at address `0xC20EF` and `publicarray_size` at address `0x0A20C0`.

6: Call `x86_get_pc_thunk_ax` and store ret addr.
2: Load `[esp]` bypasses the previous store and gets its value from main memory; which can be populated with attacker controlled values. Here, let `eax` take the transient value `0x023f35`.
7: Computes the *global pointer* for PIC using the transient value in `eax`.
8: `eax`—controlled by the attacker—is used as an offset to access the global variable `publicarray_size`. Consequently, secret data at address `0xC20EF` is loaded to `edx`.
11: Finally, the value of the secret in `edx` is used as index for a load, which violates SCT.

VII. RELATED WORK

We have discussed related work on Spectre attacks all along the paper. In this section, we discuss further the closest related work and refer the interested reader to an excellent survey by Canella et al. [2] for a more general discussion on transient execution attacks and defenses.

Speculative constant-time. Constant-time programming is often used in cryptographic code in order to prevent side-channel timing attacks [14]. Since the advent of microarchitectural attacks in 2018, a few works have extended this property to speculations [3], [12], [43]. We use in our work the property of speculative constant-time from Cauligi et al [5].

Relational symbolic execution. Relational symbolic execution [36] offers a more precise analysis than other techniques such as tainting. For instance, Pitchfork [5], which is based on tainting, reports a violation in Listing 7, line 2 because `toLeak` is tainted with secret data, whereas the program is secure because `toLeak` is set to 0 before being leaked. In contrast, BINSEC/HAUNTED, based on relational SE, does not report such false alarms.

Four previous works have used symbolic execution for analysis of cache side-channels [16], [44]–[46]. Three of them [16], [44], [45] target binary code; only two of them [16],

Tool	Technique	Target	Property	Precise	PHT	STL	Scales	Benchs
AISE [9]	Abstract Interp.	LLVM	Cache	✗	✓ NA	✗	✓	Crypto
KLEESPECTRE [11]	SE (KLEE)	LLVM	Cache	✓	Explicit*	✗	✓	Crypto
SPECUSYM [10]	SE (KLEE)	LLVM	Cache	✓	Explicit*	✗	✓	Crypto
OO7 [4]	Tainting	Binary	Patterns	✗	∼ NA	✗	✓	Other
FASS [12]	MC (UCLID5)	Binary	SNI	✓	Explicit*	✗	✗	Litmus
SPECTECTOR [3]	SE	Binary	SNI	✓	Explicit*	✗	✗	Litmus
PITCHFORK [5]	SE&taint. (ANGR)	Binary	SCT	✗	Optim-Explicit	Explicit	∼ PHT / ✗ STL	Crypto
BINSEC/HAUNTED	RelSE	Binary	SCT	✓	Haunted	Haunted	✓ PHT / ∼ STL	Crypto

Table V: Comparison of BINSEC/HAUNTED with related work where SNI denotes speculative non-interference (transient execution does not leak more information than regular execution). *These tools restrict to leaks in transient execution, so Haunted-PHT optimization does not apply, however their straightforward adaptation to SCT would be Explicit.

```
1  void leakThis(uint8_t toLeak) {
2    tmp &= publicarray2[toLeak * 512]; }
3  void case_1(uint32_t idx) {
4    if (idx < publicarray_size) {
5      uint8_t toLeak = publicarray[idx];
6      toLeak = toLeak & 0xf0;
7      toLeak = toLeak & 0x0f; // toLeak = 0
8      leakThis(toLeak); }} // Leaks value 0
```

Listing 7: Program secure to Spectre-PHT

[46] scale to real cryptographic binaries; and none of them is able to detect Spectre attacks.

Analyses for Spectre detection. Several tools have been proposed in the literature to detect Spectre vulnerabilities [3]–[5], [9]–[12] both at LLVM level and binary level. See Table V for a comparison. On one hand, analyzers at LLVM level scale well as to analyze real cryptographic code. Unfortunately, as shown in our experiments and previous works [5], [15], [16], compilers too often introduce constant-time violations. On the other hand, tools at binary level are more challenging to develop and are often ineffective on real code due to scalability issues.

These tools are based on static analysis using abstract interpretation [9], model checking [12], symbolic execution [3], [5], [10], [11] and tainting [4], [5]. However, KLEESPECTRE [11] and SPECUSYM [10] are built on top of KLEE [47] and PITCHFORK [5] on top of ANGR [48] which are dynamic symbolic execution tools and might have an additional support for concretization (but do not use it).

Four analyzers at binary level, prior to this work, constitute the state of the art [3]–[5], [12] to detect Spectre-PHT vulnerabilities but only two scale [4], [5]—by giving up on the precision (false positive). oo7 [4] relies on detecting vulnerable code pattern, while Pitchfork [5] relies on SE and taint analysis to detect secret dependent conditional statements and memory accesses.

The only previous work which addresses Spectre-STL is Pitchfork [5]. We have tested Pitchfork on our new Spectre-STL litmus tests for comparison with our work (cf. Table IV). We note that, although it is not documented in [5], Pitchfork implements an optimized exploration technique compared to Explicit for Spectre-PHT. For Spectre-STL however, it relies on Explicit and forks the execution for each transient load. Therefore, it suffers from a significant state explosion problem for Spectre-STL and quickly runs out of memory.

Currently, there is no static analyzers addressing Spectre-BTB (speculative indirect branches) or Spectre-RSB (speculative returns). While explicitly modeling transient paths underlying Spectre-BTB is in principle feasible, this is in practice intractable as it allows to jump to arbitrary addresses in the code on indirect jump instructions [5]. The same applies to Spectre-RSB, on recent Intel processors, when the return stack buffer is empty [29].

State merging in SE. State merging [21], [22] (a.k.a. path merging) is used in symbolic execution to merge states following different paths (e.g. merge diverging paths after a conditional statement). Merging in SE precisely captures the behavior of the merged states, without over-approximation: the formula of the final state the disjunction of the formula of the state to be merged. While state merging reduces the number of paths to explore, it also increases the complexity of the formula [21], consequently techniques have been proposed to selectively apply state merging [23].

For the comparison, we adopt a different strategy: we do not pack together different paths encountered along the execution, but rather prevent creating artificial path splits (unlike *Explicit*) by showing how to reason on both regular and transient executions at the same time. In our setting, a path predicate represents all input values allowing to follow a given path, be it through regular or transient executions. For PHT this is achieved through a careful handling of assertions along symbolic execution (akin to pruning), while for STL this is achieved through a symbolic encoding of memory speculations inside the path predicate (somehow akin to some merge encodings, e.g. [23], for its use of *if-then-else* expressions).

VIII. CONCLUSION

We propose Haunted RelSE, a technique built on top of relational symbolic execution to statically detect Spectre-PHT and Spectre-STL vulnerabilities. Especially, Haunted RelSE allows to significantly alleviate the cost of addressing speculative paths by reasoning about regular and transient executions at the same time. We implement Haunted RelSE in a relational symbolic execution tool, BINSEC/HAUNTED. Our experimental results show that Haunted RelSE is a step toward scalable analysis of Spectre attacks. For Spectre-PHT, Haunted RelSE can dramatically speed up the analysis in some cases,

pruning the complexity of analyzing speculative semantics on medium size real world cryptographic binaries. For Spectre-STL, BINSEC/HAUNTED is the first tool able to exhaustively analyze small real world cryptographic binaries and find vulnerabilities in medium size real world cryptographic binaries.

Finally, we report thanks to BINSEC/HAUNTED that one standard defense for Spectre-PHT can easily introduce Spectre-STL vulnerabilities, together with a fix, and also that a well-known gcc option to compile to position independent executables introduces Spectre-STL vulnerabilities.

ACKNOWLEDGMENT

The authors warmly thank the anonymous reviewers for their insightful comments, significantly helpful for improving the paper. This research was partially supported by the ANR17-CE25-0014-01 CISC project, the ANR-20-CE25-0009 TAVA project, the Inria Project Lab SPAI, and the European Union's Horizon 2020 research and innovation program under grant agreements No 830892 and No 833828.

REFERENCES

[1] P. Kocher, *Spectre Mitigations in Microsoft's C/C++ Compiler*, 2018. [Online]. Available: https : / / www . paulkocher . com / doc / MicrosoftCompilerSpectreMitigation.html.

[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] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows", in *S&P*, 2020.

[4] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "Oo7: Low-overhead Defense against Spectre Attacks via Program Analysis", *IEEE Transactions on Software Engineering*, 2020.

[5] 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.

[6] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "SpecShield: Shielding speculative data from microarchitectural covert channels", in *PACT*, 2019.

[7] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. B. Abu-Ghazaleh, "SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation", in *DAC*, 2019.

[8] *Managed Runtime Speculative Execution Side Channel Mitigations*. [Online]. Available: https://software.intel.com/security-software-guidance/deep-dives/deep-dive-managed-runtime-speculative-execution-side-channel-mitigations.

[9] M. Wu and C. Wang, "Abstract interpretation under speculative execution", in *PLDI*, 2019.

[10] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, "SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection", in *ICSE 2020 Technical Papers*, 2020.

[11] 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.*, vol. 29, no. 3, 2020.

[12] K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan, "A Formal Approach to Secure Speculation", in *CSF*, 2019.

[13] J. Horn, *Speculative execution, variant 4: Speculative store bypass*, 2018. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

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

[15] L. Simon, D. Chisnall, and R. J. Anderson, "What you get is what you C: Controlling side effects in mainstream C compilers", in *EuroS&P*, 2018.

[16] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level", in *S&P*, 2020.

[17] J. C. King, "Symbolic execution and program testing", *Commun. ACM*, vol. 19, no. 7, 1976.

[18] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: Whitebox fuzzing for security testing", *Communications of the ACM*, vol. 55, no. 3, 2012.

[19] F.Pizlo, "What spectre and meltdown mean for WebKit", 2018.

[20] F. S. Foundation, *Position Independent Code (GNU Compiler Collection (GCC) Internals)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/PIC.html.

[21] T. Hansen, P. Schachte, and H. Søndergaard, "State joining and splitting for the symbolic execution of binaries", in *RV*, 2009.

[22] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing", in *NDSS*, 2008.

[23] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution", in *PLDI*, 2012.

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

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

[26] 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.

[27] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses", *CoRR*, vol. abs/1807.03757, 2018.

[28] G. Maisuradze and C. Rossow, "Ret2spec: Speculative Execution Using Return Stack Buffers", in *CCS*, 2018.

[29] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer", in *WOOT @ USENIX Security Symposium*, 2018.

[30] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. [Online]. Available: https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html.

[31] M. R. Clarkson and F. B. Schneider, "Hyperproperties", *Journal of Computer Security*, 2010.
[32] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later", *CACM*, vol. 56, no. 2, 2013.
[33] J. Vanegue and S. Heelan, "SMT solvers in software security", in *WOOT*, 2012.
[34] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The BINCOA framework for binary code analysis", in *CAV*, 2011.
[35] *FixedSizeBitVectors Theory, SMT-LIB*. [Online]. Available: http : / / smtlib . cs . uiowa . edu / theories - FixedSizeBitVectors.shtml.
[36] G. P. Farina, S. Chong, and M. Gaboardi, "Relational symbolic execution", in *PPDP*, 2019.
[37] B. Farinier, R. David, S. Bardin, and M. Lemerre, "Arrays made simpler: An efficient, scalable and thorough preprocessing", in *LPAR*, 2018.
[38] Z. He, G. Hu, and R. B. Lee, "New models for understanding and reasoning about speculative execution attacks", *CoRR*, vol. abs/2009.07998, 2020.
[39] 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.
[40] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0 system description", *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, 2014 (published 2015).
[41] *SMT-COMP*. [Online]. Available: https : / / smt - comp . github.io/2019/results.html.
[42] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC", in *Fast Software Encryption*, 2016.
[43] M. Balliu, M. Dam, and R. Guanciale, "InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis", *CoRR*, vol. abs/1911.00868, 2019.
[44] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "CacheD: Identifying cache-based timing channels in production software", in *USENIX Security*, 2017.
[45] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. M. Fung, "Verifying information flow properties of firmware using symbolic execution", in *DATE*, 2016.
[46] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "CaSym: Cache aware symbolic execution for side channel detection and mitigation", in *IEEE Symposium on Security and Privacy*, 2019.
[47] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.", in *OSDI*, 2008.
[48] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis", in *S&P*, 2016.

APPENDIX

A. Intermediate Representation

In this section, we provide the syntax of DBA, the Intermediate Representation used by BINSEC/HAUNTED. (DBA) [34] is used by BINSEC[39] as an intermediate representation to model low-level programs and perform its analysis. The syntax of DBA programs is presented in Fig. 3. The underlying semantic is the intuitive one, with fixed-with bitvector variables (registers) and constant values, and a fixed-sized array of bytes (memory).

$$
\begin{array}{ll}
prog ::= \varepsilon \mid stmt\ prog & lval ::= \mathtt{v} \mid \mathtt{store}\ \ exp \\
stmt ::= < \mathtt{l}, inst > & exp ::= \mathtt{v} \mid \mathtt{bv} \mid \mathtt{load}\ exp \\
inst ::= lval\ \mathtt{:=}\ exp & \quad \mid \blacklozenge_u\ exp \\
\quad \mid \mathtt{ite}\ exp\mathtt{?}\ \mathtt{l}_1\mathtt{:}\mathtt{l}_2 & \quad \mid exp\ \blacklozenge_b\ exp \\
\quad \mid \mathtt{goto}\ exp \mid \mathtt{goto}\ \mathtt{l} & \blacklozenge_u ::= \neg \mid - \\
\quad \mid \mathtt{halt} & \blacklozenge_b ::= + \mid \times \mid \leq \mid \ldots
\end{array}
$$

Figure 3: DBA syntax, where $\mathtt{l}$, $\mathtt{l}_1$, $\mathtt{l}_2$ are program locations, $\mathtt{v}$ is a variable and $\mathtt{bv}$ is a value, $\blacklozenge$ (resp. $\blacklozenge_b$) represents unary (resp. binary) operators.

B. Proofs

In this section, we provide proofs for the main result of the paper. Let $C.f$ denote the field f in the symbolic configuration C. For instance, for $C \triangleq \left(l, \delta, \rho, \widehat{\mu}, \pi, \widetilde{\pi}, \widetilde{\lambda}\right)$, $C.\widetilde{\pi}$ is the speculative path predicate $\widetilde{\pi}$. Let $C_{0\ldots n}$ denote a set of n configuration $\{C_0 \ldots C_n\}$. Additionally, for a set of n sets $\{S_0 \ldots S_n\}$, $S_{0\ldots n}$ denotes the union of these sets.

Note that in Explicit RelSE, there is no speculative path predicate $\widetilde{\pi}$, nor transient load set $\widetilde{\lambda}$, but just an *invalidation depth* $\widetilde{\delta}$ at which the transient paths are terminated.

Property 1 (Unconstrained ite are sets of expressions)**.** *Let π be a path predicate and $\widehat{\varphi}$ be a symbolic if-then-else expression built over a set of relational symbolic bitvectors $\{\widehat{\varphi_1}, \ldots, \widehat{\varphi_n}\}$ and a set of booleans $\{\beta_1, \ldots, \beta_{n-1}\}$ s.t.*

$$\widehat{\varphi} \triangleq (ite\ \beta_1\ \widehat{\varphi_1}\ (\ldots (ite\ \beta_{n-1}\ \widehat{\varphi_{n-1}}\ \widehat{\varphi_n})))$$

Let $\beta_{false} \subseteq \{\beta_1, \ldots, \beta_{n-1}\}$ be the set of boolean variables that are set to $false$ in π while others are left unconstrained; and let $\widehat{\varphi}_{false} \subseteq \{\widehat{\varphi_1}, \ldots, \widehat{\varphi_{n-1}}\}$ be the corresponding set of values.

Then $\widehat{\varphi}$ can take any value in the set $\{\widehat{\varphi_1}, \ldots, \widehat{\varphi_n}\} \backslash \widehat{\varphi}_{false}$. More precisely, let $\mathbf{Bv}_i$ be the set of values (i.e. relational bitvectors) that $\widehat{\varphi_i}$ can take to satisfy π. Then $\widehat{\varphi}$ can take any value in the set $\mathbf{Bv}_{1\ldots n} \setminus \mathbf{Bv}_{false}$ to satisfy π.

Theorem 2 (Equivalence Explicit and Haunted RelSE)**.** *Haunted RelSE detects a violation in a program if and only if Explicit RelSE detects a violation.*

Proof (Sketch). Firstly we show that Theorem 2 holds for Spectre-PHT and secondly, that it holds for Spectre-STL.

Spectre-PHT. First, let us show that Theorem 2 holds for detection of Spectre-PHT vulnerabilities (we consider for this case that symbolic evaluation of loads in Haunted RelSE only returns the regular symbolic value).

Let H and E be symbolic configurations, respectively for Haunted RelSE and Explicit RelSE such that both configurations are equivalent. More precisely, their fields l, δ, ρ, $\widehat{\mu}$, and

π are equal. Additionally, consider $H.\widetilde{\pi} = \emptyset$, and $E.\widetilde{\delta} = \infty$. All the rule preserve the equivalence relation except for the rules ITE-TRUE and ITE-FALSE.

Consider that these configurations are about to execute a conditional statement $P[l] = \texttt{ite c ?}\ l_{true}\texttt{:}\ l_{false}$. Because H and E are equivalent, c evaluates to the same symbolic value (c, δ_c) in H and E. In the symbolic evaluation, both rules ITE-TRUE and ITE-FALSE can be applied.

In Haunted RelSE, evaluation forks into two paths and gives the following states:

- A state H_t following the $true$ branch s.t. $H_t.\widetilde{\pi} = \{(c, \delta_c) :: H.\widetilde{\pi}\}$ and $H_t.l = l_{true}$,
- A state H_f, following the $false$ branch s.t. $H_f.\widetilde{\pi} = \{(\neg c, \delta_c) :: H.\widetilde{\pi}\}$ and $H_f.l = l_{false}$.

In Explicit RelSE, evaluation forks into four paths and gives the following states:

- Regular $true$ state E_t s.t. $E_t.\pi' = \pi \wedge c$ and $E_t.l = l_{true}$,
- Regular $false$ state E_f s.t. $E_f.\pi' = \pi \wedge \neg c$ and $E_f.l = l_{false}$,
- Transient $true$ state E'_t s.t. $E'_t.\pi' = \pi \wedge \neg c$, $E'_t.l = l_{true}$, and $E'_t.\widetilde{\delta} = \delta_c$,
- Transient $false$ state E'_f s.t. $E'_f.\pi' = \pi \wedge c$, $E'_f.l = l_{false}$, and $E'_f.\widetilde{\delta} = \delta_c$.

We can prove by induction on the number of steps in Haunted RelSE, that there is an equivalence between Haunted RelSE and Explicit RelSE configurations:

EQ_t There is a vulnerability in execution following H_t iff there is a vulnerability in execution following E_t or in execution following E'_t.

EQ_f There is a vulnerability in execution following H_f iff there is a vulnerability in execution following E_f or in execution following E'_f.

The proof for EQ_t follows (case EQ_f is analogous):
First, we consider configurations following H_t such that the current depth δ is below the retirement depth the condition δ_c. Then, we consider configurations such that the condition is retired.

[*Case* $\delta < \delta_c$] We have to show for the rules LOAD, STORE and ITE that *secLeak* $= false$ in H_t iff it evaluates to $false$ in E_t or in E_f. Note that H_t, E_t and E_f are equivalent, thus evaluation of expressions — including symbolic leakage — is the same in the three configurations.

- Case LOAD: Let $\widehat{\varphi}$ be the symbolic value of the index. We have to show that *secLeak* $= false$ in H_t iff *secLeak* $= false$ in E_t *or* in E'_t.
  In H_t we have *secLeak* $= false$ iff $\vDash (\pi \wedge \widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$ whereas for E_t it is $\vDash (\pi \wedge c \wedge \widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$ and for E'_t it is $\vDash (\pi \wedge \neg c \wedge \widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$.
  Therefore, *secLeak* $= false$ in E_t *or* *secLeak* $= false$ in E_t iff $\vDash \big(\pi \wedge (c \wedge \widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r}) \vee (\neg c \wedge \widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})\big)$ which is equivalent to $\vDash (\pi \wedge \widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$.
  Hence, *secLeak* $= false$ in H_t iff *secLeak* $= false$ in E_t or in E'_t.
- Case STORE: Let $\widehat{\varphi}$ be the symbolic value of the index. Store instructions do not leak in transient execution, thus there is no *secLeak* check in E'_t. Therefore we have to show that *secLeak* $= false$ in H_t iff *secLeak* $= false$ in E_t.
  In H_t we first compute the regular path predicate π_{ret} by retiring c from $\widetilde{\pi}$, which gives $\pi_{ret} \triangleq \pi \wedge c$. Then we compute *secLeak* under π_{reg}, meaning that *secLeak* $= false$ iff $\vDash (\pi_{ret} \wedge \widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$
  In E_t we have *secLeak* $= false$ iff $\vDash (\pi \wedge c \wedge \widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$, which is equivalent to *secLeak* $= false$ in H_t.
  Hence *secLeak* $= false$ in H_t iff *secLeak* $= false$ in E_t.
- Case ITE: Evaluation of *secLeak* is similar as the evaluation of *secLeak* for load instructions. For symbolic states resulting from the evaluation of ITE-TRUE and ITE-FALSE, it can be shown by induction on the number of paths that the equivalence generalizes to nested conditional statements.

[*Case* $\delta \geq \delta_c$] The invalidation depth of the speculative condition c has been reached.

For Haunted RelSE, H_t evaluates the rule RETIRE-PHT, and pops the condition (c, δ_c) from $\widetilde{\pi}$, giving new symbolic state H_{ret} such that $H_{ret}.\pi = \pi \wedge c$. Note that this state is equivalent to the sate E_t.

For Explicit RelSE, $E'_t.\widetilde{\delta}$ is reached and the path E'_t is terminated, leaving only the regular state E_t.

Finally symbolic execution continues along equivalent states H_{ret} and E_t.

Spectre-STL. We show that Theorem 2 holds for detection of Spectre-STL vulnerabilities (we consider for this case that conditions cannot be mispeculated).

Let H and E be symbolic configurations, respectively for Haunted RelSE and Explicit RelSE such that both configurations are equivalent. Additionally, consider $H.\widetilde{\lambda} = \emptyset$, and $E.\widetilde{\delta} = \infty$. All the rules preserve the equivalence relation except for the rule LOAD.

Consider the evaluation of a load instruction in which the function *lookup*$_{set}$ returns a set of n values $\{(\widehat{\nu_1}, \delta_1), \ldots, (\widehat{\nu_{n-1}}, \delta_{n-1}), (\widehat{\nu_n}, \infty)\}$ such that $\widehat{\nu_n}$ is the regular value.

Explicit RelSE, forks the symbolic execution in n distinct states $E_1, \ldots, E_n$, respectively returning value $\widehat{\nu_1}, \ldots, \widehat{\nu_n}$ for the load evaluation, and such that for each $0 \leq i \leq n$ $E_i.\widetilde{\delta} = \delta_n$. Let $E_{1\ldots n}$ denote this set of configurations.

Haunted RelSE, returns a unique state H' where the load evaluates to a symbolic expression $\widehat{\nu}'$ such that:

$$\widehat{\nu}' \triangleq (ite\ \beta_1\ \widehat{\nu_1}\ (\ldots (ite\ \beta_{n-1}\ \widehat{\nu_{n-1}}\ \widehat{\nu_n})))$$

and the set of transient loads is updated as:

$$H'.\widetilde{\lambda} = \{(\beta_1, \delta_1), \ldots, (\beta_{n-1}, \delta_{n-1})\}$$

We can prove by induction on the number of steps in Haunted RelSE, that there is an equivalence between the configuration H' and the set of configurations $E_{1\ldots n}$. Namely, there is a vulnerability in the execution following H' iff there

is a vulnerability in an execution following one of the states in $E_{1\ldots n}$.

First, we consider configurations following H' where none of the transient loads have been invalidated, meaning that all β_i are unconstrained and all states E_i are still alive. Then, we consider configurations such that a transient load value must be invalidated (i.e. there is δ_i such that the current depth δ is greater or equal than δ_i).

[*Case* $\delta < \delta_i$] We have to show for the rules LOAD, STORE and ITE that *secLeak* $= false$ in H' iff it evaluates to $false$ in one of the states in $E_{1\ldots n}$.

- Case LOAD: We have to show that *secLeak* $= false$ in H' iff *secLeak* $= false$ in one of the states in $E_{1\ldots n}$.
  Let $\widehat{\varphi'}$ be the symbolic value of the index in H', and for all $1 \leq i \leq n$, let $\widehat{\varphi_i}$ be the symbolic value of the index in $E_i \in E_{1\ldots n}$.
  In Explicit RelSE, for $E_i \in E_{1\ldots n}$, we have *secLeak* $= false$ iff $\vDash \pi_i \wedge \widehat{\varphi_i}_{|l} \neq \widehat{\varphi_i}_{|r}$. Let $\mathbf{Bv}_i$ be the set of values that $\widehat{\varphi_i}$ can take to satisfy π_i. We have *secLeak* $= false$ in one of the state in $E_{1\ldots n}$ iff there is $\widehat{bv} \in \mathbf{Bv}_{1\ldots n}$ such that $\widehat{bv}_{|l} \neq \widehat{bv}_{|r}$.
  In Haunted RelSE, the symbolic index $\widehat{\varphi'}$ can take different values according to the value of $\widehat{\nu'}$. Because all β_i are still unconstrained variables, from Property 1, we have $\widehat{\varphi'}$ can take any value in $\{\widehat{\varphi_1} \ldots \widehat{\varphi_n}\}$. Therefore, to satisfy $H.\pi$, $\widehat{\varphi}$ can take any value in $\mathbf{Bv}_{1\ldots n}$. Therefore, we have *secLeak* $= false$ in H' iff there is $\widehat{bv} \in \mathbf{Bv}_{1\ldots n}$ such that $\widehat{bv}_{|l} \neq \widehat{bv}_{|r}$.
  Hence, *secLeak* $= false$ in H' iff *secLeak* $= false$ in one of the states in $E_{1\ldots n}$.
- Case STORE: Let $\widehat{\varphi}$ be the symbolic value of the index. Store instructions do not leak in transient execution so there is no *secLeak* check in $E_{1\ldots n-1}$. Therefore we have to show that *secLeak* $= false$ in H' iff *secLeak* $= false$ in E_n.
  In Explicit RelSE, for E_n, we have *secLeak* $= false$ iff $\vDash \pi_n \wedge \widehat{\varphi_n}_{|l} \neq \widehat{\varphi_n}_{|r}$. Let $\mathbf{Bv}_n$ be the set of values that $\widehat{\varphi_n}$ can take to satisfy π_n. We have *secLeak* $= false$ in E_n iff there is $\widehat{bv} \in \mathbf{Bv}_n$ such that $\widehat{bv}_{|l} \neq \widehat{bv}_{|r}$.
  In Haunted RelSE, for H', the rule first computes the regular path predicate π_{reg} by invalidating all transient loads in the transient load set $H'.\widetilde{\lambda}$, giving $\pi_{reg} \triangleq H.\pi \wedge \neg\beta_1 \wedge \cdots \wedge \neg\beta_{n-1}$. Note that under this constraint, from Property 1, we have $\widehat{\varphi} = \widehat{\varphi_n}$ and thus $\widehat{\varphi}$ can take any value in $\mathbf{Bv}_n$ to satisfy $H.\pi$. Finally *secLeak* $= false$ in H iff there is $\widehat{bv} \in \mathbf{Bv}_n$ such that $\widehat{bv}_{|l} \neq \widehat{bv}_{|r}$, which is equivalent to *secLeak* $= false$ in E_n.
  Hence *secLeak* $= false$ in H' iff *secLeak* $= false$ in E_n.
- Case ITE: Evaluation of *secLeak* is similar as the evaluation of *secLeak* for load instructions.

[*Case* $\delta \geq \delta_i$] The invalidation depth of the transient value $\widehat{\nu_i}$ has been reached.

For Haunted RelSE, H' evaluates the rule RETIRE-STL. The rule invalidates the transient value ν_i by removing the boolean β_i from $\widetilde{\lambda}$ and setting it to false in the path predicate. This gives the new symbolic state H_{ret} such that $H_{ret}.\pi = \pi \wedge \neg\beta$. From Property 1, this restricts the set of possible value for $\widehat{\nu'}$ to $\{\nu_0 \ldots \nu_n\} \setminus \{\nu_i\}$.

For Explicit RelSE, $E_i.\widetilde{\delta}$ is reached and the path E_i is terminated. This restricts the set of possible states to $E_{1\ldots n} \setminus \{E_i\}$.

Finally symbolic execution continues along equivalent states H_{ret} and $E_{1\ldots n} \setminus \{E_i\}$, and in both cases ν can take any value in $\{\nu_0 \ldots \nu_n\} \setminus \{\nu_i\}$. □

C. Haunted-PHT on programs containing loops.

We illustrate on litmus test `case_5` (Listing 8) the importance of Haunted RelSE for path pruning in programs containing loops. In this program the loop is bounded by the size of the array and can be fully unrolled in in-order execution. In transient execution, the loop can be mispeculated but unrolling is eventually bounded by the speculation depth. Performance of Explicit RelSE and Haunted RelSE are reported in Table VI.

```
1  void case_5(uint64_t idx) {
2    int64_t i;
3    if (idx < publicarray_size)
4      for (i = idx - 1; i >= 0; i--)
5        temp &= publicarray2[publicarray[i]*512];
6  }
```

Listing 8: Litmus `case_5` where `publicarray_size` is set to 16.

PHT	UInstr.	Paths	Time (s)	
NoSpec	305	17	1.3	0
Explicit	6824	407	26.5	1
Haunted	589	32	1.9	1

Table VI: Comparison of Explicit and Haunted RelSE for Spectre-PHT on litmus `case_5` where UInstr is the number of unrolled `x86` instructions.

RelSE restricted to in-order execution (NoSpec) produces 17 paths: a first path exits after the conditional at line 3, and 16 different path come from unrolling the loop 0 to 15 times.

Explicit RelSE forks into four paths after the conditional branch at line 3, two of them jumping on the loop at line 4. Then, each time the condition of the loop is evaluated, Explicit RelSE forks again into four paths[6]. In total, 390 additional transient paths are explored (Table VI). The behavior of Haunted RelSE, is close to NoSpec: it only forks into two paths at line 3 and when the condition of the loop is evaluated. However, while NoSpec stops after 15 iterations of the loop, Haunted RelSE transiently executes the loop up to 15 times[7], which gives a total of 32 paths.

This example illustrates how Haunted RelSE can prune redundant paths compared to Explicit RelSE, achieving performance closer to standard (in-order) RelSE. *Haunted RelSE spares 375 paths compared to Explicit RelSE and is almost 14 times faster.*

[6]Depending on the path predicate, either the four paths are satisfiable or only two of them are satisfiable.

[7]The loop body is 14 instructions long and can therefore be speculatively executed 15 times in a speculation window of 200 instructions.