BINSEC/REL: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level

Lesly-Ann Daniel*, Sébastien Bardin*, Tamara Rezk†

* CEA, List, Université Paris-Saclay, France
† INRIA Sophia-Antipolis, INDES Project, France

lesly-ann.daniel@cea.fr, sebastien.bardin@cea.fr, tamara.rezk@inria.fr

***Abstract*—The constant-time programming discipline (CT) is an efficient countermeasure against timing side-channel attacks, requiring the control flow and the memory accesses to be independent from the secrets. Yet, writing CT code is challenging as it demands to reason about pairs of execution traces (2-hypersafety property) and it is generally not preserved by the compiler, requiring binary-level analysis. Unfortunately, current verification tools for CT either reason at higher level (C or LLVM), or sacrifice bug-finding or bounded-verification, or do not scale. We tackle the problem of designing an efficient binary-level verification tool for CT providing both bug-finding and bounded-verification. The technique builds on relational symbolic execution enhanced with new optimizations dedicated to information flow and binary-level analysis, yielding a dramatic improvement over prior work based on symbolic execution. We implement a prototype, BINSEC/REL, and perform extensive experiments on a set of 338 cryptographic implementations, demonstrating the benefits of our approach in both bug-finding and bounded-verification. Using BINSEC/REL, we also automate a previous manual study of CT preservation by compilers. Interestingly, we discovered that `gcc -O0` and backend passes of `clang` introduce violations of CT in implementations that were previously deemed secure by a state-of-the-art CT verification tool operating at LLVM level, showing the importance of reasoning at binary-level.**

I. INTRODUCTION

Timing channels occur when timing variations in a sequence of events depends on secret data. They can be exploited by an attacker to recover secret information such as plaintext data or secret keys. *Timing attacks*, unlike other side-channel attacks (e.g based on power-analysis, electromagnetic radiation or acoustic emanations) do not require special equipment and can be performed remotely [1], [2]. First timing attacks exploited secret-dependent *control flow* with measurable timing differences to recover secret keys [3] from cryptosystems. With the increase of shared architectures (e.g. infrastructure as a service) arise more powerful attacks where an attacker can monitor the cache of the victim and recover information on secret-dependent *memory accesses* [4]–[6].

Therefore, it is of paramount importance to implement adequate countermeasures to protect cryptographic implementations from these attacks. Simple countermeasures consisting in adding noise or dummy computations can reduce timing variations and make attacks more complex. Yet, these mitigations eventually become vulnerable to new generations of attacks and provide only *pseudo* security [7].

The *constant-time* programming discipline (CT) [8], a.k.a. constant-time policy, is a software-based countermeasure to timing attacks which requires the control flow and the memory accesses of the program to be independent from the secret input[1]. Constant-time has been proven to protect against cache-based timing attacks [8], making it the most effective countermeasure against timing attacks, already widely used to secure cryptographic implementations (e.g. BearSSL [9], NaCL [10], HACL* [11], etc).

Problem. Writing constant-time code is complex as it requires low-level operations deviating from traditional programming behaviors. Moreover, this effort is brittle as it is generally not preserved by compilers [12], [13]. For example, reasoning about CT requires to know whether the code `c=(x<y)-1` will be compiled to branchless code, but this depends on the compiler version and optimization [12]. As shown in the attack on a "constant-time" implementation of elliptic curve Curve25519 [13], writing constant-time code is error prone [7], [12], [13].

Several CT-analysis tools have been proposed to analyze source code [14], [15], or LLVM code [16], [17], but leave the gap opened for violations introduced in the executable code either by the compiler [12] or by closed-source libraries [13]. Binary-level tools for CT using dynamic aproaches [18]–[21] can find bugs but miss vulnerabilities in unexplored portions of the code, making them incomplete; while static approaches [22]–[24] cannot report precise counterexamples – making them of minor interest when the implementation cannot be proven secure. Aside from a posteriori analysis, correct-by-design approaches [11], [25]–[27] require to reimplement cryptographic primitives from scratch, and OS-based countermeasures [28]–[31] incur runtime overhead and require specific OS- or hardware-support.

Challenges. Two main challenges arise in the verification of constant-time:

- First, common verification methods do not directly apply because information flow properties like CT are not

[1] Some versions of CT also require that the size of operands of variable-time instructions (e.g. integer division) is independent from secrets.

- regular safety properties but 2-hypersafety properties [32] (i.e. relating two execution traces), and their standard reduction to safety on a transformed program, *self-composition* [33], is inefficient [34];
- Second, it is notoriously difficult to adapt formal methods to binary-level because of the lack of structure information (data & control) and the explicit representation of the memory as a single large array of bytes [35], [36].

A technique that scales well on binary code and that naturally comes into play for bug-finding and bounded-verification is *symbolic execution* (SE) [37], [38]. While it has proven very successful for standard safety properties [39], its direct adaptation to CT and other 2-hypersafety properties through (variants of) self-composition suffers from a scalability issue [40]–[42]. Some recent approaches achieve better scaling, but at the cost of sacrificing either bounded-verification [20], [43] (under-approximation) or bug-finding [17] (over-approximations).

The idea of analyzing pairs of executions for the verification of 2-hypersafety is not new (e.g. relational Hoare logic [44], self-composition [33], product programs [45], multiple facets [46], [47]). In the context of symbolic execution, it has first been coined as *ShadowSE* [48] for back-to-back testing, and later as *relational symbolic execution* (RelSE) [49]. However, a direct adaptation of this technique *does not scale in the context of binary-level analysis* because of the representation of the memory as a single large array which prevents sharing between executions, sending *a high number of queries* to the constraint solver which could have been simplified beforehand with a better information flow tracking.

Proposal. *We tackle the problem of designing an efficient symbolic verification tool for constant-time at binary-level that leverages the full power of symbolic execution without sacrificing correct bug-finding nor bounded-verification.* We present BINSEC/REL, the first efficient binary-level automatic tool for bug-finding and bounded-verification of constant-time at binary-level. It is compiler-agnostic, targets x86 and ARM architectures and does not require source code.

The technique is based on *relational symbolic execution* [48], [49]. It models two execution traces following the same path in the same symbolic execution instance and *maximizes sharing between them*. We show via experiments (Section VII-B) that RelSE (and ShadowSE) alone does not scale at binary-level to analyze CT on real cryptographic implementations.

Our key technical insights are (1) to use this sharing to track secret-dependencies and reduce the number of queries sent to the solver, and (2) to complement it with dedicated optimizations offering a fine-grained information flow tracking in the memory for efficient binary analysis.

BINSEC/REL can analyze about 23 million instructions in 98 min, (i.e. 3860 instructions per second), outperforming similar state of the art binary-level verification tools based on symbolic execution [20], [43] (cf. Table VIII, page 13), while being still correct and complete for CT.

Contributions. Our contributions are the following:

- We design dedicated optimizations for information flow analysis at binary-level. First, we complement relational symbolic execution with a new *on-the-fly* simplification for *binary-level* analysis, to track secret-dependencies and maximize sharing in the memory (Section V-A1). Second, we design new simplifications for *information flow* analysis: untainting (Section V-A2) and fault-packing (Section V-A3). Moreover, we formally prove that our analysis is correct for bug-finding and bounded-verification of CT (Section V-B).
- We propose a verification tool named BINSEC/REL for CT analysis (Section VI). Extensive experimental evaluation (338 samples) against standard approaches based on self-composition and RelSE (Section VII-B) shows that it can find bugs in real-world cryptographic implementations much faster than these techniques ($\times 700$ speedup) and can achieve bounded-verification when they timeout, with performances close to standard SE ($\times 1.8$ overhead).
- In order to prove the effectiveness of BINSEC/REL, we perform an extensive analysis of CT at binary-level. In particular, we analyze 296 cryptographic implementations previously verified at a higher-level (incl. codes from HACL* [11], BearSSL [9], NaCL [10]), we replay known bugs in 42 samples (incl. Lucky13 [50]) and automatically generate counterexamples (Section VII-A).
- Simon *et al.* [12] have demonstrated that `clang`'s optimizations break constant-timeness of code. We extend this work in four directions – from 192 in [12] to 408 configurations (Section VII-A): (1) we automatically analyze the code that was manually checked in [12], (2) we add new implementations, (3) we add the `gcc` compiler and a more recent version of `clang`, (4) we add ARM binaries. Interestingly, we discovered that `gcc -O0` and backend passes of `clang` with `-O3 -m32 -march=i386` introduce violations of CT that cannot be detected by LLVM verification tools like ct-verif [16].

Our technique is shown to be highly efficient on bug-finding and bounded-verification compared to alternative approaches, paving the way to a systematic binary-level analysis of CT on cryptographic implementations, while our experiments demonstrate the importance of developing CT-verification tools reasoning at binary-level. Besides CT, the technique can be readily adapted to other hyperproperties of interest in security (e.g., cache-side channels, secret-erasure), as long as they are restricted to pairs of traces following the same path.

II. BACKGROUND

In this section, we present the basics of contant-time and symbolic execution. Small examples of CT and standard adaptations of symbolic execution are presented in Section III, while a formal definition of CT is given in Section IV.

A. Constant-Time

Information flow policies regulate the transfer of information between public and secret domains. To reason about information flow, we partition the program input into two disjoint sets: *low* (i.e. public) and *high* (i.e. secret). Typical information flow properties require that the observable output of a program does not depend on the high input (*non-interference*). CT is a special case requiring both the program control flow and the memory accesses to be independent from high input.

Contrary to a standard *safety* property which states that nothing bad can happen along *one execution trace*, information flow properties relate *two execution traces* – they are *2-hypersafety properties* [32]. Unfortunately, the vast majority of symbolic execution tools [37], [51]–[55] is designed for safety verification and cannot directly be applied to 2-hypersafety properties. In principle, 2-hypersafety properties can be reduced to standard safety properties of a *self-composed program* [33] but this reduction alone does not scale [34].

B. Symbolic Execution

Symbolic Execution (SE) [37], [38], [56] consists in executing a program on *symbolic inputs* instead of concrete input values. Variables and expressions of the program are represented as terms over symbolic inputs and the current path is modeled by a *path predicate* (a logical formula), which is the conjunction of conditional expressions encountered along the execution.

SE is built upon two main steps. (1) *Path search*: at each conditional statement the symbolic execution *forks*, the expression of the condition is added to the first branch and its negation to the second branch, then the symbolic execution continues along satisfiable branches; (2) *Constraint solving*: the path predicate can be solved with an off-the-shelf *automated constraint solver*, typically SMT [57], to generate a concrete input exercising the path.

Combining these two steps, SE can explore many different program paths and generate test inputs exercising these paths. It can also check local assertions in order to *find bugs* or perform *bounded-verification* (i.e., verification up to a certain depth). Dramatic progresses in program analysis and constraint solving over the last two decades have made SE a tool of choice for intensive testing [38], [39], vulnerability analysis [52], [58], [59] and other security-related analysis [60], [61].

C. Binary-Level Symbolic Execution

Low-level code operates on a set of registers and a single (large) untyped memory. During the execution, a call stack contains information about the active functions such as their arguments and local variables. A special register `esp` (stack pointer) indicates the top address of the call stack and local variables of a function can be referenced as offsets from the initial `esp`[2].

Binary-level symbolic execution. Binary-level code analysis is notoriously more challenging than source code analysis [35], [36]. First, evaluation and assignments of source code variables become memory load and store operations, requiring to reason explicitely about the memory in a very precise way. Second, the high level control flow structure (e.g. `for` loops) is not preserved, and there are dynamic jumps to handle (e.g. instruction of the form `jmp eax`).

Fortunately, it turns out that SE is less difficult to adapt from source code to binary code than other semantic analysis – due to both the efficiency of SMT solvers and concretization (i.e., simplifying a formula by constraining some variables to be equal to their observed runtime values). Hence, strong binary-level SE tools do exist and have yielded several highly promising case studies [37], [52]–[55], [61], [62].

Logical notations. Binary-level SE relies on the theory of bitvectors and arrays, QF_ABV [63]. Values (e.g. registers, memory addresses, memory content) are modeled with fixed-size bitvectors [64]. We will use the type $\mathcal{B}v_m$, where m is a constant number, to represent symbolic bitvector expressions. The memory is modeled with a logical array [65], [66] of type $(Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)$ (assuming a 32 bit architecture). A logical array is a function $(Array\ \mathcal{I}\ \mathcal{V})$ that maps each index $i \in \mathcal{I}$ to a value $v \in \mathcal{V}$. Operations over arrays are:

- $select : (Array\ \mathcal{I}\ \mathcal{V}) \times \mathcal{I} \rightarrow \mathcal{V}$ takes an array a and an index i and returns the 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.

These functions satisfy the following constraints for all $a \in (Array\ \mathcal{I}\ \mathcal{V})$, $i \in \mathcal{I}$, $j \in \mathcal{I}$, $v \in \mathcal{V}$:

- $select\ (store\ a\ i\ v)\ i = v$
- $i \neq j \implies select\ (store\ a\ i\ v)\ j = select\ a\ j$

III. Motivating Example

Let us consider the toy program in Listing 1. The value of the conditional at line 3 and the memory access at line 4 are *leaked*. We say that a *leak is insecure* if it depends on the secret input. Conversely, a *leak is secure* if it does not depend on the secret input. CT holds for a program if there is no insecure leak.

```
1  x := private_input();
2  y := public_input();
3  if y then return 0;  // leak y = 0
4  else return tab[x];  // leak x
```

Listing 1: Toy program with one control-flow leak and one memory leak.

Let us take two executions of this program with the same public input: (x, y) and (x', y') where $y = y'$. Intuitively, we can see that the leakages produced at line 3, $y = 0$ and $y' = 0$, are necessarily equal in both executions because $y = y'$; hence this leak does not depend on the secret input and is secure. On the contrary, we can see that the leakages x and x' at line 4 can differ in both executions (e.g. take $x := 0$ and $x' := 1$); hence this leak depends on the secret input and is insecure.

The goal of an automatic analysis is to prove that the leak at line 3 is secure and to return concrete input showing that the leak at line 4 is insecure.

[2] `esp` is specific to x86, but this is generalizable, e.g. `sp` for ARMv7.

Symbolic Execution & Self-Composition (SC). Symbolic execution can be adapted to the case of CT following the self-composition principle. Instead of self-composing the program, we rather self-compose the formula with a renamed version of itself plus a precondition stating that the low inputs are equal. Basically, this amounts to model *two different executions following the same path and sharing the same low input* in a single formula. At each conditional statement, *exploration queries* are sent to the solver to determine satisfiable branches – followed by both executions (similar to standard SE exploration). Moreover, additional *insecurity queries* specific to CT are sent before each conditional statement and memory access to determine if they depend on the secret or not – if an insecurity query is satisfiable then a CT violation is found.

As an illustration, let us consider the program in Listing 1. First, we assign symbolic values to `x` and `y` and use symbolic execution to generate a formula of the program until the first conditional (line 3), resulting in the formula: $x = \beta \ \wedge \ y = \lambda \ \wedge \ c = (\lambda > 0)$. Second, self-composition is applied on the formula with precondition $\lambda = \lambda'$ to constraint the low inputs to be equal in both executions. Finally, a postcondition $c \neq c'$ asks whether the value of the conditional can differ, resulting in the following insecurity query:

$$\lambda = \lambda' \ \wedge \ \begin{pmatrix} x = \beta \ \wedge \ y = \lambda \ \wedge \ c = (\lambda > 0) \ \wedge \\ x' = \beta' \ \wedge \ y' = \lambda' \ \wedge \ c' = (\lambda' > 0) \end{pmatrix} \ \wedge \ c \neq c'$$

This formula is sent to a SMT-solver. If the solver returns UNSAT, meaning that the query is not satisfiable, then the conditional does not differ in both executions and thus is secure. Otherwise, it means that the outcome of the conditional depends on the secret and the solver will return a counterexample satisfying the insecurity query. Here, `z3` answers that the query is UNSAT and we can conclude that the leak is secure. With the same method, the analysis finds that the leak at line 4 is insecure, and returns two inputs (0,0) and (1,0), respectively leaking 0 and 1, as a counterexample showing the violation.

Limits. Basic self-composition suffer from two weaknesses:

- It generates lots of insecurity queries – at each conditional statement and memory access. Yet, in the previous example it is clear that the conditional does not depend on secrets and could be spared with better information flow tracking.
- The whole original formula is duplicated so the size of the self-composed formula is twice the size of the original formula. Yet, because the parts of the program that only depend on public inputs are equal in both executions, the self-composed formula contains redundancies that are not exploited.

Relational Symbolic Execution (RelSE). We can improve SC by maximizing *sharing* between the pairs of executions [48], [49]. As previously, RelSE models two executions of a program P in the same symbolic execution instance, let us call them p and p'. But in RelSE, variables of P are mapped to *relational expressions* which are either *pairs* of expressions or *simple* expressions. The variables that *must be equal* in p and p' (typically, the low inputs) are represented as *simple* expressions while those that *may be different* are represented as *pairs* of expressions. First, this enables to share redundant parts of p and p', reducing the size of the self-composed formula. Second, variables mapping to simple expressions cannot depend on the secret input, allowing to easily spare some insecurity queries.

As an example, let us perform RelSE of the toy program in Listing 1. Variable `x` is assigned a pair of expressions $\langle \beta \,|\, \beta' \rangle$ and `y` is assigned a simple expression $\langle \lambda \rangle$. Note that the precondition that public variables are equal is now implicit since we use the same symbolic variable in both executions. At line 3, the conditional expression is evaluated to $c = \langle \lambda > 0 \rangle$ and we need to check that the leakage of c is secure. Since c maps to a simple expression, we know by definition that it does not depend on the secret, hence we can spare the insecurity query.

RelSE maximizes sharing between both executions and tracks secret-dependencies enabling to spare insecurity queries and reduce the size of the formula.

Challenge of binary-level analysis. Recall that, in binary-level SE, the memory is represented as a special variable of type $(Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)$. We cannot directly store relational expressions in it, so in order to store high inputs at the beginning of the execution, we have to duplicate it. In other words the *memory is always duplicated*. Consequently, every $select$ operation will evaluate to a duplicated expression, preventing to spare queries in many situations.

As an illustration, consider the compiled version of the previous program, given in Listing 2. The steps of RelSE on this program are given in Fig. 1. Note that when the secret input is stored in memory at line (1), the array representing the memory is duplicated. This propagates to the load expression in `eax` at line (3) and to the conditional expression at line (4). Intuitively, at line (4), `eax` should be equal to the simple expression $\langle \lambda \rangle$ in which case we could spare the insecurity query like in the previous example. However, because dependencies cannot be tracked in the array representing the memory, `eax` evaluates to a pair of $select$ expression and we have to send the insecurity query to the solver.

```
1 @[ebp-8] := ⟨β|β'⟩; // store high input
2 @[ebp-4] := ⟨λ⟩;    // store low input
3 eax := @[ebp-4];    // assign ⟨λ⟩ to eax
4 ite eax ? l1 : l2;  // leak ⟨λ⟩
5 [...]
```

Listing 2: Compiled version of the conditional in Listing 1, where `x` := $\langle \beta \mid \beta' \rangle$ (resp. `x` := $\langle \lambda \rangle$) denotes that `x` is assigned a high (resp. low) input.

Practical impact. We report in Table I the performances of CT-analysis on an implementation of elliptic curve Curve25519-donna [67]. *Both SC and RelSE fail to prove the program secure in less than 1h. RelSE* does reduce the number of queries w.r.t. *SC*, but it is not sufficient.

$$
\begin{array}{rll}
(\mathit{init}) & \texttt{mem} := \langle \mu_0 \rangle \text{ and } \texttt{ebp} := \langle \mathit{ebp} \rangle & \\
(1) & \texttt{mem} := \langle \mu_1 \,|\, \mu_1' \rangle & \text{where } \mu_1 \triangleq \mathit{store}(\mu_0, \mathit{ebp}-8, \beta) \\
 & & \text{and } \mu_1' \triangleq \mathit{store}(\mu_0, \mathit{ebp}-8, \beta') \\
(2) & \texttt{mem} := \langle \mu_2 \,|\, \mu_2' \rangle & \text{where } \mu_2 \triangleq \mathit{store}(\mu_1, \mathit{ebp}-4, \lambda) \\
 & & \text{and } \mu_2' \triangleq \mathit{store}(\mu_1', \mathit{ebp}-4, \lambda) \\
(3) & \texttt{eax} := \langle \alpha \,|\, \alpha' \rangle & \text{where } \alpha \triangleq \mathit{select}(\mu_2, \mathit{ebp}-4) \\
 & & \text{and } \alpha' \triangleq \mathit{select}(\mu_2', \mathit{ebp}-4) \\
(4) & \mathit{leak}\ \langle \alpha \neq 0 \,|\, \alpha' \neq 0 \rangle &
\end{array}
$$

Figure 1: RelSE of program in Listing 2 where `mem` is the memory variable, `ebp` and `eax` are registers, $\mu_0, \mu_1, \mu_1', \mu_2, \mu_2'$ are symbolic array variables, and $\mathit{ebp}, \beta, \beta', \lambda, \alpha, \alpha'$ are symbolic bitvector variables

Version	#I	#Q	T	#I/s	S
SC (e.g. [43])	11k	9051	TO	3	⌛
RelSE (e.g. [49])	13k	5486	TO	4	⌛
BINSEC/REL	10M	0	1166	8576	✓

Table I: Performances of CT-analysis of `donna` compiled with `gcc-5.4 -O0`, in terms of number of explored unrolled instructions (#I), number of queries (#Q), execution time in seconds (T), instructions explored per second (#I/s), and status (S) set to secure (✓) or timeout (⌛) set to 3600s.

Our solution. To mitigate this issue, we propose dedicated simplifications for binary-level relational symbolic execution that allow a precise tracking of secret-dependencies *in the memory* (details in Section V-A). In the particular example of Table I, our prototype BINSEC/REL *does prove that the code is secure* in less than 20 minutes. Our simplifications simplify *all the queries*, resulting in a $\times 2000$ speedup compared to standard RelSE and SC in terms of number of instructions treated per second.

IV. CONCRETE SEMANTICS & FAULT MODEL

Dynamic Bitvectors Automatas (DBA) [68] is used by BINSEC[55] as an Intermediate Representation to model low-level programs and perform its analysis. The syntax of DBA programs is presented in Fig. 2.

$$
\begin{array}{rcl|rcl}
\mathit{prog} & ::= & \varepsilon \mid \mathit{stmt}\ \mathit{prog} & \mathit{lval} & ::= & \texttt{v} \mid \texttt{@}\ \mathit{expr} \\
\mathit{stmt} & ::= & <\texttt{l}, \mathit{inst}> & \mathit{expr} & ::= & \texttt{v} \mid \texttt{bv} \mid \texttt{@}\ \mathit{expr} \\
\mathit{inst} & ::= & \mathit{lval} \ \texttt{:=}\ \mathit{expr} & & & \mid \blacklozenge_u\ \mathit{expr} \\
 & & \mid \texttt{ite}\ \mathit{expr}\texttt{?}\ \texttt{l}_1\texttt{:}\texttt{l}_2 & & & \mid \mathit{expr}\ \blacklozenge_b\ \mathit{expr} \\
 & & \mid \texttt{goto}\ \mathit{expr} \mid \texttt{goto l} & \blacklozenge_u & ::= & \neg \mid - \\
 & & \mid \texttt{halt} & \blacklozenge_b & ::= & + \mid \times \mid \leq \mid \ldots
\end{array}
$$

Figure 2: The syntax of DBA programs, where `l`, $\texttt{l}_1$, $\texttt{l}_2$ are program locations, `v` is a variable and `bv` is a value.

Let Inst denote the set of instructions and Loc the set of (program) locations. A program $P : \mathit{Loc} \to \mathit{Inst}$ is a map from locations to instructions. Values `bv` and variables `v` range over the set of fixed-size bitvectors $BV_n := \{0,1\}^n$ (set of n-bit words). A concrete configuration is a tuple (l, r, m) where:

- $l \in \mathit{Loc}$ is the current location, and $P.l$ returns the current instruction,
- $r : \mathit{Var} \to BV_n$ is a register map that maps variables to their bitvector value,
- $m : BV_{32} \to BV_8$ is the memory, mapping 32-bit addresses to bytes and is accessed by the operator @ (read in an expression and write in a left value).

The initial configuration is given by $c_0 \triangleq (l_0, r_0, m_0)$ with l_0 the address of the entrypoint of the program, r_0 an arbitrary register map, and m_0 an arbitrary memory.

Leakage model. The behavior of the program is modeled with an instrumented operational semantics taken from [69] in which each transition is labeled with an explicit notion of leakage. A transition from a configuration c to a configuration c' produces a leakage t, denoted $c \underset{t}{\to} c'$. Analogously, the evaluation of an expression e in a configuration (l, r, m), denoted $(l, r, m)\ e \vdash_t bv$, produces a leakage t. The leakage of a multistep execution is the concatenation of leakages produced by individual steps. We use $\underset{t}{\to}^k$ with k a natural number to denote k steps in the concrete semantics.

An excerpt of the concrete semantics is given in Fig. 3 where leakage by memory accesses occur during execution of load and store instructions and control flow leakages during execution of dynamic jumps and conditionals. The full set of rules is given in Appendix A1.

$$
\textsc{load}\ \frac{(l,r,m)\ e \vdash_t \texttt{bv}}{(l,r,m)\ \texttt{@}\ e \vdash_{t\cdot[\texttt{bv}]} m\ \texttt{bv}}
$$

$$
\textsc{d\_jump}\ \frac{P.l = \texttt{goto}\ e \qquad (l,r,m)\ e \vdash_t \texttt{bv} \qquad l' \triangleq \mathit{to\_loc}(\texttt{bv})}{(l,r,m) \xrightarrow[t\cdot[l']]{} (l',r,m)}
$$

$$
\textsc{t-ite}\ \frac{P.l = \texttt{ite}\ e\ \texttt{?}\ l_1\texttt{:}\ l_2 \qquad (l,r,m)\ e \vdash_t \texttt{bv} \qquad \texttt{bv} \neq 0}{(l,r,m) \xrightarrow[t\cdot[l_1]]{} (l_1,r,m)}
$$

$$
\textsc{store}\ \frac{\begin{array}{c} P.l = \texttt{@}e := e' \\ (l,r,m)\ e \vdash_t \texttt{bv} \qquad (l,r,m)\ e' \vdash_{t'} \texttt{bv}' \end{array}}{(l,r,m) \xrightarrow[t'\cdot t\cdot[\texttt{bv}]]{} (l+1, r, m[\texttt{bv} \mapsto \texttt{bv}'])}
$$

Figure 3: Concrete evaluation of DBA instructions and expressions (excerpt), where $\cdot$ is the concatenation of leakages and $\mathit{to\_loc} : BV_{32} \to \mathit{Loc}$ converts a bitvector to a location.

Secure program. Let $H_v \subseteq \mathit{Var}$ be the set of high (secret) variables and $L_v := \mathit{Var} \setminus H_v$ be the set of low (public) variables. Analogously, we define $H_@ \subseteq BV_{32}$ (resp. $L_@ := BV_{32} \setminus H_@$) as the addresses containing high (resp. low) input in the initial memory.

The *low-equivalence relation* over concrete configurations c and c', denoted $c \simeq_L c'$, is defined as the equality of low variables and low parts of the memory. Formally, two configurations $c \triangleq (l, r, m)$ and $c' \triangleq (l', r', m')$ are low-equivalent iff,

$$\forall v \in L_v.\ r\ v = r'\ v$$
$$\forall a \in L_@.\ m\ a = m'\ a$$

Definition 1 (Constant-time up to k)**.** *A program is constant-time (CT) up to k iff for all low-equivalent initial configurations c_0 and c_0', that evaluate in k steps to c_k and c_k' producing leakages t and t',*

$$c_0 \simeq_L c_0' \quad \wedge \quad c_0 \xrightarrow[t]{}^k c_k \quad \wedge \quad c_0' \xrightarrow[t']{}^k c_k' \quad \Longrightarrow \quad t = t'$$

V. Binary-level Relational Symbolic Execution

Our symbolic execution relies on the QF_ABV [63] first-order logic. We let β, β', λ, φ, i, j range over the set of formulas Φ in the QF_ABV logic. A *relational* formula $\widehat{\varphi}$ is either a QF_ABV formula $\langle\varphi\rangle$ or a pair $\langle\varphi_l \mid \varphi_r\rangle$ of two QF_ABV formulas. We denote $\widehat{\varphi}_{|l}$ (resp. $\widehat{\varphi}_{|r}$), the projection on the left (resp. right) value of $\widehat{\varphi}$. If $\widehat{\varphi} = \langle\varphi\rangle$, then $\widehat{\varphi}_{|l}$ and $\widehat{\varphi}_{|r}$ are both defined as φ. We let $\mathbf{\Phi}$ be the set of relational formulas and $\mathcal{B}\mathbf{v_n}$ be the set of relational symbolic bitvectors of size n.

Symbolic configuration. Since we restrict our analysis to pairs of traces following the same path – which is sufficient for constant-time – the symbolic configuration only considers a single program location $l \in Loc$ at any point of the execution.

A *symbolic configuration* is of the form $(l, \rho, \widehat{\mu}, \pi)$ where:

- $l \in Loc$ is the current program point,
- $\rho : Var \to \mathbf{\Phi}$ is a symbolic register map, mapping variables from a set Var to their symbolic representation as a relational formula in $\mathbf{\Phi}$,
- $\widehat{\mu} : (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8) \times (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)$ is the symbolic memory – a pair of arrays of values in $\mathcal{B}v_8$ indexed by addresses in $\mathcal{B}v_{32}$,
- $\pi \in \Phi$ is the path predicate – a conjunction of conditional statements and assignments encountered along a path.

Symbolic evaluation of instructions, denoted $s \rightsquigarrow s'$ where s and s' are symbolic configurations, is given in Figure 4 – the complete set of rules is given in Appendix A2. The evaluation of an expression in a state $(\rho, \widehat{\mu})$ to a relational formula $\widehat{\varphi}$, is given by $(l, \rho, \widehat{\mu})\ expr \vdash \widehat{\varphi}$. A model M assigns concrete values to symbolic variables. The satisfiability of a formula π with a model M is denoted $M \vDash \pi$. Whenever the model is not needed for our purposes, we leave it implicit and simply write $\vDash \pi$ for satisfiability. In the implementation, we use an SMT-solver to determine satisfiability of a formula.

For the security evaluation of the symbolic leakage we define a function $secLeak$ which verifies that a relational formula in the symbolic leakage does not differ in its right and left components, i.e. that the symbolic leakage is secure:

$$secLeak\,(\widehat{\varphi}) = \begin{cases} true & \text{if } \widehat{\varphi} = \langle\varphi\rangle \\ true & \text{if } \widehat{\varphi} = \langle\varphi_l \mid \varphi_r\rangle \wedge \nvDash \left(\pi \wedge (\varphi_l \neq \varphi_r\right) \\ false & \text{otherwise} \end{cases}$$

Notice that a simple expression $\langle\varphi\rangle$ does not depend on secrets and can be leaked securely. Thus it *spares an insecurity query* to the solver. On the other hand, a duplicated expression $\langle\varphi_l \mid \varphi_r\rangle$ *may* depend on secrets. Hence *an insecurity query must be sent to the solver* to ensure that the leak is secure.

Detailed explanations of the symbolic evaluation rules follow:

LOAD is the evaluation of a load expression. The rule returns a pair of logical *select* formulas from the pair of symbolic memories $\widehat{\mu}$ (the box in the hypotheses should be ignored for now, it will be explained in Section V-A). Note that the returned expression is *always duplicated* as the *select* must be performed in the left and right memories independently.

D_JUMP is the evaluation of a dynamic jump. The rule finds a concrete value l' for the jump target, and updates the path predicate and the location. Note that this rule is nondeterministic as l' can be any concrete value satisfying the constraint. In practice, we call the solver to enumerate jump targets up to a given bound and continue the execution along the valid targets (which jump to an executable section).

ITE-TRUE is the evaluation of a conditional jump when the expression evaluates to true (the false case is analogous). The rule updates the path predicate and the next location accordingly.

STORE is the evaluation of a store instruction. The rule evaluates the index and value of the store and updates the symbolic memories and the path predicate with a logical *store* operation.

LOAD

$$\frac{(\rho, \widehat{\mu})\ e \vdash \widehat{\phi} \qquad \boxed{\widehat{\varphi} \triangleq \langle select(\widehat{\mu}_{|l}, \widehat{\phi}_{|l}) \mid select(\widehat{\mu}_{|r}, \widehat{\phi}_{|r})\rangle} \qquad secLeak\,(\widehat{\phi})}{(\rho, \widehat{\mu})\ @e \vdash \widehat{\varphi}}$$

D_JUMP

$$\frac{P.l = \texttt{goto}\ e \qquad (\rho, \widehat{\mu})\ e \vdash \widehat{\varphi} \qquad \pi' \triangleq \pi \wedge (\widehat{\varphi}_{|l} = \widehat{\varphi}_{|r}) \qquad M \vDash \pi' \qquad l' \triangleq M(\widehat{\varphi}_{|l}) \qquad secLeak\,(\widehat{\varphi})}{(l, \rho, \widehat{\mu}, \pi) \rightsquigarrow (l', \rho, \widehat{\mu}, \pi')}$$

ITE-TRUE

$$\frac{P.l = \texttt{ite}\ e\ \texttt{?}\ l_{true}\texttt{:}\ l_{false} \qquad l' \triangleq l_{true} \qquad (\rho, \widehat{\mu})\ e \vdash \widehat{\varphi} \qquad \pi' \triangleq \pi \wedge (true = \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r}) \qquad \vDash \pi' \qquad secLeak\,(\widehat{\varphi})}{(l, \rho, \widehat{\mu}, \pi) \rightsquigarrow (l', \rho, \widehat{\mu}, \pi')}$$

STORE

$$\frac{\begin{array}{c} P.l = @e := e' \\ l' = l + 1 \qquad (\rho, \widehat{\mu})\ e \vdash \widehat{\varphi} \qquad (\rho, \widehat{\mu})\ e' \vdash \widehat{\phi} \\ \widehat{\mu}' \triangleq \langle store(\widehat{\mu}_{|l}, \widehat{\varphi}_{|l}, \widehat{\phi}_{|l}) \mid store(\widehat{\mu}_{|r}, \widehat{\varphi}_{|r}, \widehat{\phi}_{|r})\rangle \\ \pi' \triangleq \pi \wedge \widehat{\mu}'_{|l} = store(\widehat{\mu}_{|l}, \widehat{\varphi}_{|l}, \widehat{\phi}_{|l}) \wedge \widehat{\mu}'_{|r} = store(\widehat{\mu}_{|r}, \widehat{\varphi}_{|r}, \widehat{\phi}_{|r}) \\ secLeak\,(\widehat{\varphi}) \end{array}}{(l, \rho, \widehat{\mu}, \pi) \rightsquigarrow (l', \rho, \widehat{\mu}', \pi')}$$

Figure 4: Symbolic evaluation of DBA instructions and expressions (excerpt).

Specification of high and low input. By default, the content of the memory and registers is low so we have to specify addresses that initially contain secret inputs. The addresses of high variables can be specified as offsets from the initial stack pointer `esp`. A pair $\langle \beta \,|\, \beta' \rangle \in \mathcal{B}\mathbf{v_8}$ of fresh symbolic variables is stored at each given offset h and modifies the symbolic configuration just as a store instruction `@[esp + h]` $:= \langle \beta \,|\, \beta' \rangle$ would. Similarly, offsets containing low inputs can be set to simple symbolic expressions $\langle \lambda \rangle$ – although it is not necessary since the initial memory is equal in both executions.

Bug-Finding. A vulnerability is found when the function $secLeak\,(\varphi)$ evaluates to *false*. In this case, the insecurity query is satisfiable and there exists a model M such that $M \vDash \pi \wedge (\widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$. The model M assigns concrete values to variables that satisfy the insecurity query. Therefore it can be returned as a concrete counterexample which triggers the vulnerability, along with the current location l of the vulnerability.

A. Optimizations for binary-level SE

Relational symbolic execution does not scale in the context of binary-level analysis (see *RelSE* in Table V). In order to achieve better scalability, we enrich our analysis with an optimization, called *on-the-fly-read-over-write* (*FlyRow* in Table VI), based on *read-over-write* [66]. This optimization simplifies expressions and resolves load operations ahead of the solver, often avoiding to resort to the duplicated memory and allowing to spare insecurity queries. We also enrich our analysis with two further optimizations, called *untainting* and *fault-packing* (*Unt* and *fp* in Table VI), specifically targeting SE for information flow analysis.

1) On-the-Fly Read-Over-Write: Solver calls are the main bottleneck of symbolic execution, and reasoning about $store$ and $select$ operations in arrays is particularly challenging [66]. Read-over-write (Row) [66] is a simplification for the theory of arrays that efficiently resolves $select$ operations. This simplification is particularly efficient in the context of binary-level analysis because the memory is represented as an array and formulas contain many $store$ and $select$ operations.

The standard read-over-write optimization [66] has been implemented as a solver-pre-processing, simplifying a formula before sending it to the solver. While it has proven to be very efficient to simplify individual formulas of a single execution [66], we show in Section VII-B that it does not scale in the context of relational reasoning, where formulas model two executions and a lot of queries are sent to the solver.

Thereby, we introduce *on-the-fly read-over-write* (*FlyRow*) to track secret-dependencies in the memory and spare insecurity queries in the context of information flow analysis. By keeping track of *relational* $store$ expressions along the symbolic execution, it can resolve $select$ operations – often avoiding to resort to the duplicated memory – and drastically reduces the number of queries sent to the solver, improving the performances of the analysis.

Lookup. The symbolic memory can be seen as the history of the successive $store$ operations beginning with the initial memory μ_0. Therefore, a memory $select$ can be resolved by going back up the history and comparing the index to load, with indexes previously stored. Our optimization consists in replacing selection in the memory (Figure 4, LOAD rule, boxed hypothesis) by a new function $lookup : ((Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8) \times (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)) \times \mathcal{B}v_{32} \rightarrow \mathcal{B}\mathbf{v_8}$ which takes a relational memory and an index, and returns the relational value stored at that index. The lookup function can be lifted to relational indexes but for simplicity we only define it for simple indexes and assume that relational store operations happen to the same index in both sides – note that for constant-time analysis, this hypothesis holds. The function returns a relational bitvector formula, and is defined as follows:

$$lookup\,(\widehat{\mu}_0, i) = \quad \langle select(\widehat{\mu}_{0|l}, i) \,|\, select(\widehat{\mu}_{0|r}, i) \rangle$$

$$lookup\,(\widehat{\mu}_n, i) =$$

$$\begin{cases} \langle \varphi_l \rangle & \text{if } eq^{\#}(i,j) \wedge eq^{\#}(\varphi_l, \varphi_r) \\ \langle \varphi_l \,|\, \varphi_r \rangle & \text{if } eq^{\#}(i,j) \wedge \neg eq^{\#}(\varphi_l, \varphi_r) \\ lookup\,(\widehat{\mu}_{n-1}, i) & \text{if } \neg eq^{\#}(i,j) \\ \widehat{\phi} & \text{if } eq^{\#}(i,j) = \bot \end{cases}$$

where

$$\widehat{\mu}_n \triangleq \langle store(\widehat{\mu}_{n-1|l}, j, \varphi_l) \,|\, store(\widehat{\mu}_{n-1|r}, j, \varphi_r) \rangle$$

$$\widehat{\phi} \triangleq \langle select(\widehat{\mu}_{n|l}, i) \,|\, select(\widehat{\mu}_{n|r}, i) \rangle$$

where $eq^{\#}(i,j)$ is a comparison function relying on *syntactic term equality*, which returns true (resp. false) only if i and j are equal (resp. different) in any interpretation. If the terms are not comparable, it is undefined, denoted $\bot$.

Example 1 (Lookup). Let us consider the memory:

$$\widehat{\mu} \quad = \quad \boxed{ebp - 4 \mid \langle \lambda \rangle} \longrightarrow \boxed{ebp - 8 \mid \langle \beta \,|\, \beta' \rangle} \longrightarrow \boxed{esp \mid \langle ebp \rangle} \longrightarrow [\ ]$$

- A call to $lookup\,(\widehat{\mu}, ebp - 4)$ returns λ.
- A call to $lookup\,(\widehat{\mu}, ebp - 8)$ first compares the indexes $[ebp{-}4]$ and $[ebp{-}8]$. Because it can determine that these indexes are *syntactically distinct*, the function moves to the second element, determines the syntactic equality of indexes and returns $\langle \beta \,|\, \beta' \rangle$.
- A call to $lookup\,(\widehat{\mu}, esp)$ tries to compare the indexes $[ebp{-}4]$ and $[esp]$. Without further information, the equality or disequality of ebp and esp cannot be determined, therefore the lookup is aborted and the $select$ operation cannot be simplified.

Term rewriting. To improve the conclusiveness of this syntactic comparison, the terms are assumed to be in *normalized* form $\beta + o$ where β is a base (i.e. an expression on symbolic variables) and o is a constant offset. In order to apply *FlyRow*, we normalize all the formulas created during the symbolic execution (details of our normalization function are ommited for space reasons). The comparison of two terms $\beta + o$ and $\beta' + o'$ in normalized form can be efficiently computed as

$$untaint(\rho, \widehat{\mu}, \langle v_l \,|\, v_r \rangle) = (\rho[v_r \backslash v_l], \widehat{\mu}[v_r \backslash v_l])$$

$$\left.\begin{array}{l} untaint(\rho, \widehat{\mu}, \langle \neg t_l \,|\, \neg t_r \rangle) \\ untaint(\rho, \widehat{\mu}, \langle -t_l \,|\, -t_r \rangle) \\ untaint(\rho, \widehat{\mu}, \langle t_l + k \,|\, t_r + k \rangle) \\ untaint(\rho, \widehat{\mu}, \langle t_l - k \,|\, t_r - k \rangle) \\ untaint(\rho, \widehat{\mu}, \langle t_l :: k \,|\, t_r :: k \rangle) \end{array}\right\} = untaint(\rho, \widehat{\mu}, \langle t_l \,|\, t_r \rangle)$$

Figure 5: Untainting rules where v_l, v_r are bitvector variables and t_l, t_r, k are arbitrary bitvector terms, and $f[v_r \backslash v_l]$ indicates that the variable v_r is substituted with v_l in f.

follows: if the bases β and β' are syntactically equal, then return $o = o'$, otherwise the terms are not comparable.

In order to increase the conclusiveness of *FlyRow*, we also need variable inlining. However, inlining all variables is not a viable option as it would lead to an exponential term size growth. Instead, we define a *canonical form* $v + o$ where v is a bitvector variable, and o is a constant bitvector offset, and we only inline formulas that are in canonical form. It enables rewriting of most of the memory accesses on the stack which are of the form `ebp + bv` while avoiding term-size explosion.

2) Untainting: After the evaluation of a rule with the predicate $secLeak$ for a duplicated expression $\langle \varphi_l \,|\, \varphi_r \rangle$, we know that the equality $\varphi_l = \varphi_r$ holds in the current configuration. From this equality, we can deduce useful information about variables that must be equal in both executions. We can then propagate this information to the register map and memory in order to spare subsequent insecurity queries concerning these variables. For instance, consider the leak of the duplicated expression $\langle v_l + 1 \mid v_r + 1 \rangle$, where v_l and v_r are symbolic variables. If the leak is secure, we can deduce that $v_l = v_r$ and replace all occurrences of v_r by v_l in the rest of the symbolic execution.

We define a function $untaint\,(\rho, \widehat{\mu}, \widehat{\varphi})$ that takes a register map ρ, a memory $\widehat{\mu}$, and a duplicated expression $\widehat{\varphi}$; it applies the rules defined in Fig. 5 which deduce variable equalities from $\widehat{\varphi}$, propagate them in ρ and $\widehat{\mu}$, and return a pair of updated register map and memory $(\rho', \widehat{\mu}')$. Intuitively, if the equality of variables v_l and v_r can be deduced from $secLeak\,(\widehat{\varphi})$, the $untaint$ function replaces occurences of v_r by v_l in the memory and the register map. As a result, a duplicated expression $\langle v_l \,|\, v_r \rangle$ would be replaced by the simple expression $\langle v_l \rangle$ in the rest of the execution[3].

3) Fault-Packing: For CT, the number of insecurity checks generated along the symbolic execution is substantial. The fault-packing (*fp*) optimization gathers these insecurity checks along a path and postpones their resolution to the end of the basic block.

Example 2 (Fault-packing). For example, let us consider a basic-block with a path predicated π. If there are two memory accesses along the basic block that evaluate to $\langle \varphi \mid \varphi' \rangle$ and $\langle \phi \mid \phi' \rangle$, we would normally generate two insecurity queries $(\pi \wedge \varphi \neq \varphi')$ and $(\pi \wedge \phi \neq \phi')$ – one for each memory access. *fp* regroups these checks into a single query $\big(\pi \wedge ((\varphi \neq \varphi') \vee (\phi \neq \phi'))\big)$ sent to the solver at the end of the basic block.

This optimization reduces the number of insecurity queries sent to the solver and thus helps improving performance. However it degrades the precision of the counterexample: while checking each instruction individually precisely points to vulnerable instructions, fault-packing reduces accuracy to vulnerable basic blocks only. Note that even though disjunctive constraints are usually harder to solve than pure conjunctive constraints, those introduced by *fp* are very limited (no nesting) and thus do not add much complexity. Accordingly, they never end up in a performance degradation (see Table VI).

B. Theorems

In order to define properties of our symbolic execution, we use $\rightarrow^k$ (resp. $\rightsquigarrow^k$), with k a natural number, to denote k steps in the concrete (resp. symbolic) evaluation.

Definition 2 ($\approxeq_p^M$)**.** *We define a concretization relation $\approxeq_p^M$ between concrete and symbolic configurations, where M is a model and $p \in \{l, r\}$ is a projection on the left or right side of a symbolic configuration. Intuitively, the relation $c \approxeq_p^M s$ is the concretization of the p-side of the symbolic state s with the model M. Let $c \triangleq (l_1, r, m)$ and $s \triangleq \big(l_2, \rho, \widehat{\mu}, \pi\big)$. Formally $c \approxeq_p^M s$ holds iff $M \vDash \pi$, $l_1 = l_2$ and for all expression e, either the symbolic evaluation of e gets stuck or we have*

$$\big(\rho, \widehat{\mu}\big)\ e \vdash \widehat{\varphi}\ \wedge\ (M(\widehat{\varphi}_{|p}) = \mathtt{bv} \iff c\ e \vdash \mathtt{bv})$$

Notice that because both executions represented in the initial configuration s_0 are low-equivalent, $c_0 \approxeq_l^M s_0 \wedge c_0' \approxeq_r^M s_0$ implies that $c_0 \simeq_L c_0'$.

Through this section, we assume that the program P is defined on all locations computed during the symbolic execution. Under this hypothesis, the symbolic execution can only get stuck when an expression $\widehat{\varphi}$ is leaked such that $\neg secLeak\,(\widehat{\varphi})$. In this case, a vulnerability is detected and there exists a model M such that $M \vDash \pi \wedge (\widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$.

The following theorem claims the completeness of our symbolic execution relatively to an initial symbolic state. If the program is constant-time up to k, then for each pair of concrete executions up to k, there exists a corresponding symbolic execution (no under-approximation). A *proof* is given in Appendix B1.

Theorem 1 (Relative Completeness of RelSE)**.** *Let P be a program constant-time up to k and s_0 be a symbolic initial configuration for P. For every concrete states c_0, c_k, c_0', c_k', and model M such that $c_0 \approxeq_l^M s_0\ \wedge\ c_0' \approxeq_r^M s_0$, if $c_0 \underset{t}{\rightarrow}^k c_k$ and $c_0' \underset{t'}{\rightarrow}^k c_k'$ with $t = t'$ then there exists a symbolic configuration s_k and a model M' such that:*

$$s_0 \rightsquigarrow^k s_k\ \wedge\ c_k \approxeq_l^{M'} s_k\ \wedge\ c_k' \approxeq_r^{M'} s_k$$

[3]We implement untainting with a cache of "untainted variables" that are substituted in the program copy when relational expressions are built.

The following theorem claims the correctness of our symbolic execution, stating that for each symbolic execution and model M satisfying the path predicate, the concretization of the symbolic execution with M corresponds to a valid concrete execution (no over-approximation). A *proof* is given in Appendix B2.

Theorem 2 (Correctness of RelSE)**.** *For every symbolic configurations s_0, s_k such that $s_0 \rightsquigarrow^k s_k$ and for every concrete configurations c_0, c_k and model M, such that $c_0 \approxeq_p^M s_0$ and $c_k \approxeq_p^M s_k$, there exists a concrete execution $c_0 \rightarrow^k c_k$.*

The following is our main result. If the symbolic execution does not get stuck due to a satisfiable insecurity query, then the program is constant-time. The *proof* is given in Appendix B3.

Theorem 3 (Bounded-Verification for CT)**.** *Let s_0 be a symbolic initial configuration for a program P. If the symbolic evaluation does not get stuck, then P is constant-time w.r.t. s_0. Formally, if for all k, $s_0 \rightsquigarrow^k s_k$ then for all initial configurations c_0 and c_0' and model M such that $c_0 \approxeq_l^M s_0$, and $c_0' \approxeq_r^M s_0$,*

$$c_0 \simeq_L c_0' \;\wedge\; c_0 \xrightarrow[t]{}^k c_k \;\wedge\; c_0' \xrightarrow[t']{}^k c_k' \implies t = t'$$

Additionally, if s_0 is fully symbolic, then P is constant-time.

The following theorem expresses that when the symbolic execution gets stuck, then there is a concrete path that violates constant-time. The *proof* is given in Appendix B4.

Theorem 4 (Bug-Finding for CT)**.** *Let s_0 be an initial symbolic configuration for a program P. If the symbolic evaluation gets stuck in a configuration s_k then P is not constant-time. Formally, if there exists k st. $s_0 \rightsquigarrow^k s_k$ and s_k is stuck, then there exists a model M and concrete configurations $c_0 \approxeq_l^M s_0$, $c_0' \approxeq_r^M s_0$, $c_k \approxeq_l^M s_k$ and $c_k' \approxeq_r^M s_k$ such that,*

$$c_0 \simeq_L c_0' \;\wedge\; c_0 \xrightarrow[t]{}^k c_k \;\wedge\; c_0' \xrightarrow[t']{}^k c_k' \wedge t \neq t'$$

VI. Implementation

We implemented our relational symbolic execution, BINSEC/REL, on top of the binary-level analyzer BINSEC [55]. BINSEC/REL takes as input a x86 or ARM executable, a specification of high inputs and an initial memory configuration (possibly fully symbolic). It performs bounded exploration of the program under analysis (up to a user-given depth), and reports the identified CT violations together with counterexemples (i.e., initial configurations leading to the vulnerabilities). In case no violation is reported, if the initial configuration is fully symbolic and the program has been explored exhaustively then the program is *proven* secure.

BINSEC/REL is composed of a *relational symbolic exploration* module and an *insecurity analysis* module. The symbolic exploration module chooses the path to explore, updates the symbolic configuration, builds the path predicate and ensure that it is satisfiable. The insecurity analysis module builds insecurity queries and check that they are not satisfiable.

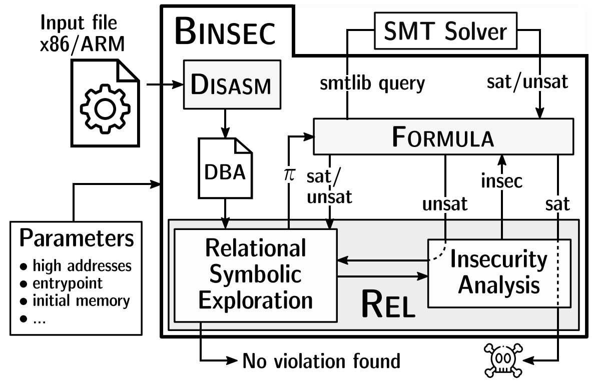

Figure 6: BINSEC architecture with BINSEC/REL plugin.

We explore the program in a depth-first search manner and we rely on the Boolector SMT-solver [70], currently the best on theory QF_ABV [66], [71].

Overall architecture is illustrated in Fig. 6. The DISASM module loads the executable and lifts the machine code to the DBA intermediate representation [68]. Then, the analysis is performed by the REL module on the DBA code. The FORMULA module is in charge of building and simplifying formulas, and sending the queries to the SMT-solver. The queries are exported to the SMTLib [63] standard which permits to interface with many off-the-shelf SMT-solvers. The REL plugin represents ≈3.5k lines of Ocaml.

Usability. Binary-level semantic analyzers tend to be harder to use than their source-level counterparts as inputs are more difficult to specify and results more difficult to interpret. In order to mitigate this point, we propose a vizualisation mechanism (based on IDA, which highlight coverage and violations) and easy input specification (using dummy functions, cf. Appendix C) when source-level information is available.

VII. Experimental Results

We answer the following research questions:

- **RQ1: Effectiveness** Is BINSEC/REL able to perform constant-time analysis on real cryptographic binaries, for both bug finding and bounded-verification?
- **RQ2: Genericity** Is BINSEC/REL generic enough to encompass several architectures and compilers?
- **RQ3: Comparison vs. Standard Approaches** How does BINSEC/REL scale compared to traditional approaches based on standard SC and RelSE?
- **RQ4: Impact of simplifications** What are the respective impacts of our different simplifications?
- **RQ5: Comparison vs. SE** What is the overhead of BINSEC/REL compared to standard SE, and can our simplifications be useful for standard SE?

Experiments were performed on a laptop with an Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz processor and 32GB of RAM, running Linux Mint 18.3 Sylvia. Similarly to related work (e.g. [23]), `esp` is initialized to a concrete value, we start the analysis from the beginning of the `main` function, we statically allocate data structures and the length of keys and buffers is fixed (e.g. for Curve25519-donna [67], three 256-bit buffers are used to store the input, the output and the secret key). When not stated otherwise, programs are compiled for x86 (32bit) with their default compiler setup.

A. Effectiveness (RQ1,RQ2)

We carry out three experiments to assess the effectiveness of our technique: (1) bounded-verification of secure cryptographic primitives previously verified at source- or LLVM-level [11], [15], [16], (2) automatic replay of known bug studies [12], [16], [50], (3) automatic study of CT preservation by compilers extending prior work [12]. Overall, our study encompasses 338 representative code samples for a total of 70k machine instructions and 22M unrolled instructions (i.e., instructions explored by BINSEC/REL).

Bounded-Verification (RQ1). We analyze a large range of *secure* constant-time cryptographic primitives (296 samples, 64k instructions), comprising: (1) several basic constant-time utility functions such as selection functions [12], sort functions [72] and utility functions from HACL*[4] and OpenSSL[5]; (2) a set of representative constant-time cryptographic primitives already studied in the literature on source code [15] or LLVM [16], including implementations of TEA [73], Curve25519-donna [67], `aes` and `des` encryption functions taken from BearSSL [9], cryptographic primitives from libsodium [10] and the constant-time padding remove function `tls-cbc-remove-padding` from OpenSSL [16]; (3) a set of functions from the HACL* library [11].

Results are reported in Table II. For each program, BINSEC/REL is able to perform an exhaustive exploration without finding any violations of constant-time in less than 20 minutes. Note that exhaustive exploration is possible because in cryptographic programs, bounding the input size bounds loops. These results show that BINSEC/REL can perform bounded-verification of real-world cryptographic implementations at binary-level in a reasonable time, which was impractical with previous approaches based on self-composition or standard RelSE (see Section VII-B).

This is the first automatic CT-analysis of these cryptographic libraries at the binary-level.

Bug-Finding (RQ1). We take three known bug studies from the literature [12], [50], [72] and replay them automatically at binary-level (42 samples, 6k instructions), including: (1) binaries compiled from constant-time sources of a selection function [12] and sort functions [72], (2) non-constant-time versions of `aes` and `des` from BearSSL [9], (3) the non-constant-time version of OpenSSL's `tls-cbc-remove-padding`[6] responsible for the famous Lucky13 attack [50].

[4]https://github.com/project-everest/hacl-star/blob/master/snapshots/hacl-c/Hacl_Policies.c and https://github.com/project-everest/hacl-star/blob/master/snapshots/hacl-c/kremlib_base.h

[5]https://github.com/xbmc/openssl/blob/master/crypto/constant_time_locl.h

[6]https://github.com/openssl/openssl/blob/OpenSSL_1_0_1/ssl/d1_enc.c

		≈ #I	#I$_u$	T	S
utility	ct-select	1015	1507	.21	29 × ✓
	ct-sort	2400	1782	.24	12 × ✓
	Hacl*	3850	90953	9.34	110 × ✓
	OpenSSL	4550	5113	.75	130 × ✓
tea	-O0	290	953	.12	✓
	-O3	250	804	.12	✓
donna	-O0	7083	10.2M	1166	✓
	-O3	4643	2.7M	401	✓
libsodium	salsa20	1627	6.5k	.7	✓
	chacha20	2717	30.0k	5.0	✓
	sha256	4879	38.7k	4.5	✓
	sha512	16312	62.1k	7.1	✓
Hacl*	chacha20	1221	5.0k	1.0	✓
	curve25519	8522	9.4M	1110	✓
	sha256	1279	16.8k	2.8	✓
	sha512	2013	31.8k	4.3	✓
BearSSL	aes_ct	357	3.5k	.6	✓
	des_ct	682	38.5k	33.9	✓
OpenSSL	tls-rempad-patch	424	35.7k	406	✓
Total		64114	22.7M	3154	296 × ✓

Table II: Bounded verification of constant-time cryptographic implementations where #I (resp. #I$_u$) is the number of static (resp. unrolled) instructions, T is the execution time in seconds, and S is the status (⌛ for timeout or ✓ for exhaustive exploration).

Results are reported in Table III with *fault-packing disabled* to report vulnerabilities at the instruction level. All bugs have been found within the timeout. Interestingly, we found 3 *unexpected binary-level vulnerabilities (from secure source codes) that slipped through previous analysis*:

- function `ct_select_v1` [12] was deemed secured through binary-level manual inspection, still we confirm that any version of `clang` with `-O3` introduces a secret-dependent conditional jump which violates constant-time;
- functions `ct_sort` and `ct_sort_mult`, verified by ct-verif [16] (LLVM bytecode compiled with `clang`), are vulnerable when compiled with `gcc -O0` or `clang -O3 -m32 -march=i386`.

A few more details on these vulnerabilities are provided in the next study. Finally, we describe the application of BINSEC/REL to the Lucky13 attack in Appendix D.

		≈ #I	#I$_u$	T	CT src	S	🐞	Comment
utility	ct-select	735	767	.29	Y	21×✗	21	1 new ✗
	ct-sort	3600	7513	13.3	Y	18×✗	44	2 new ✗
BearSSL	aes_big	375	873	1574	N	✗	32	-
	des_tab	365	10421	9.4	N	✗	8	-
OpenSSL	tls-rempad-luk13	950	11372	2574	N	✗	5	-
Total		6025	30946	4172	-	42×✗	110	-

Table III: Bug-finding of constant-time in cryptographic implementations where #I (resp. #I$_u$) is the number of static (resp. unrolled) instructions, T is the execution time in seconds, CT src means that the source is constant-time, S is the status (✗ for insecure program), and 🐞 is the number of bugs.

Effects of compiler optimizations on CT (RQ1, RQ2). Simon *et al.* [12] *manually* analyse whether `clang` optimizations break the constant-time property, for 5 different versions of a selection function. We reproduce their analysis in an *automatic* manner and *extend it significantly*, adding: 29 new functions, a newer version of `clang`, the ARM architecture, the `gcc` compiler and `arm-linux-gnueabi-gcc` version 5.4.0 for ARM – for a total of 408 executables (192 in the initial study). Results are presented in Table IV.

	cl-3.0		cl-3.9		**cl-7.1**		**gcc-5.4**		**gcc-8.3**		**arm-gcc**	
	`O0`	`O3`	`O0`	`O3`	`O0`	`O3`	`O0`	`O3`	`O0`	`O3`	`O0`	`O3`
ct_select_v1	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v2	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v3	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v4	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
select_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
ct_sort	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
ct_sort_mult	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
sort_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
hacl_utility (×11)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
openssl_utility (×13)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
tea_encrypt	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
tea_decrypt	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table IV: Constant-time analysis of several functions compiled with `gcc` or `clang` (cl) and optimization levels `O0` or `O3`. ✓ indicate that the program is secure and ✗ that it is insecure. **Bold programs** and **compilers** are extensions of [12] and ✗ indicates a different result than [12].

We *confirm* the main conclusion of Simon *et al.* [12] that `clang` is more likely to optimize away CT protections as the optimization level increases. Yet, *contrary to their work*, our experiments show that newer versions of `clang` are not necessarily more likely than older ones to break CT (e.g. `ct_sort` is compiled to a non-constant-time code with `clang-3.9` but not with `clang-7.1`).

Surprisingly, in contrast with `clang`, `gcc` optimizations tend to remove branches and thus, are less likely to introduce vulnerabilities in constant-time code. Especially, `gcc` *for ARM produces secure binaries from the insecure source codes* [7] `sort_naive` and `select_naive`.

Although [12] reports that the `ct_select_v1` function is secure in all their settings, we find the opposite. Manual inspection confirms that `clang` with `-O3` introduces a secret-dependent conditional jump violating constant-time.

Finally, as previously discussed, we found that the `ct_sort` and `ct_sort_mult` functions, taken from the benchmark of the `ct-verif` [16] tool, can be compiled to insecure binaries. Those vulnerabilities are out of reach of `ct-verif` because it targets LLVM code compiled with `clang`, while the vulnerabilities are either introduced by `gcc` or by *backend passes* of `clang` – we did confirm that `ct-verif` with the setting `-clang-options="-O3 -m32 -march=i386"` does not report the vulnerability.

Conclusion (RQ1, RQ2). We perform an extensive analysis over 338 samples of representative cryptographic primitive studied in the literature [11], [15], [16], compiled with different versions and options of `clang` and `gcc`, over x86 and ARM. Overall, it demonstrates that BINSEC/REL does scale to realistic applications for both bug-finding and bounded-verification (RQ1), and that the technology is generic (RQ2). We also get the following interesting side results:

- We proved CT-secure 296 binaries of interest;
- We found 3 new vulnerabilities that slipped through previous analysis – manual on binary code [12] or automated on LLVM [16];
- We significantly extend and automate a previous study on effects of compilers on CT [12];
- We found that `gcc` optimizations tend to help enforcing CT – on ARM, `gcc` even sometimes produces secure binaries from insecure sources.

B. Comparisons (RQ3,RQ4,RQ5)

We compare BINSEC/REL with standard approaches based on self-composition (*SC*) and relational symbolic execution (*RelSE*) (RQ3), then we analyze the performances of our different simplifications (RQ4), and finally we investigate the overhead of BINSEC/REL compared to standard SE, and whether our simplifications are useful for SE (RQ5).

Experiments are performed on the programs introduced in Section VII-A for bug-finding and bounded-verification (338 samples, 70k instructions). We report the following metrics: total number of unrolled instruction #I, number of instruction explored per seconds (#I/s), total number of queries sent to the solver (#Q), number of exploration (resp. insecurity) queries (#Q_e), (resp. #Q_i), total execution time (T), timeouts (⌛), programs proven secure (✓), programs proven insecure (✗), unknown status (~). Timeout is set to 3600 seconds.

Comparison vs. Standard Approaches (RQ3). We evaluate BINSEC/REL against *SC* and *RelSE*. Since no implementation of these methods fit our particular use-cases, we implement them directly in BINSEC. *RelSE* is obtained by disabling BINSEC/REL optimizations (Section V-A), while *SC* is implemented on top of *RelSE* by duplicating low inputs instead of sharing them and adding the adequate preconditions. Results are given in Table V.

	#I	#I/s	#Q	#Q_e	#Q_i	T	⌛	✓	✗	~
SC	252k	3.9	170k	16k	154k	65473	15	282	41	15
RelSE	320k	5.4	97k	19k	78k	59316	14	283	42	13
BINSEC/REL	22.8M	3861	3.9k	2.7k	1.3k	5895	0	296	42	0

Table V: BINSEC/REL vs. standard approaches

While *RelSE* performs slightly better than *SC* (×1.38 speedup) thanks to a noticeable reduction of the number of queries (≈50%), both techniques are not efficient enough on binary code: *RelSE* times out in 14 cases and achieves an analysis speed of only 5.4 instructions per second while *SC* is worse. BINSEC/REL *completely outperforms both previous approaches*, especially its simplifications drastically reduce the number of queries sent to the solver (×60 less insecurity queries than *RelSE*):

- BINSEC/REL reports no timeout, it is 715 times faster than *RelSE* and 1000 times faster than *SC*;

[7]The compiler takes advantage of the many ARM conditional instructions to remove conditional jumps.

Version	#I	#I/s	#Q	$\#Q_e$	$\#Q_i$	T	⧗	✓	✗	~
Standard RelSE with *Unt* and *fp*										
RelSE	320k	5.4	96919	19058	77861	59316	14	283	42	13
+ *Unt*	373k	8.4	48071	20929	27142	44195	8	288	42	8
+ *fp*	391k	10.5	33929	21649	12280	37372	7	289	42	7
BINSEC/REL (*RelSE + FlyRow + Unt + fp*)										
RelSE+FlyRow	22.8M	3075	4018	2688	1330	7402	0	296	42	0
+ *Unt*	22.8M	3078	4018	2688	1330	7395	0	296	42	0
+ *fp*	22.8M	3861	3980	2688	1292	5895	0	296	42	0

Table VI: Performances of BINSEC/REL simplifications.

- BINSEC/REL performs bounded-verification of large programs (e.g. `donna`, `des_ct`, `chacha20`, etc.) that were out of reach of standard methods.

Performances of Simplifications (RQ4). We consider on-the-fly read-over-write (*FlyRow*), untainting (*Unt*) and fault-packing (*fp*). Results are reported in Table VI:

- *FlyRow* is the major source of improvement in BINSEC/REL, drastically reducing the number of queries sent to the solver and allowing a $\times 569$ speedup w.r.t. *RelSE*;
- Untainting and fault-packing do have a positive impact on *RelSE* (untainting alone reduces the number of queries by 50%, the two optimizations together yield a $\times 2$ speedup);
- Yet, their impact is more modest once *FlyRow* is activated: untainting leads to a very slight speedup, while fault-packing still achieves a $\times 1.25$ speedup.

Still, *fp* can be interesting on some particular programs, when the precision of the bug report is not the priority. Consider for instance the non-constant-time version of `aes` in BearSSL (i.e. `aes_big`): BINSEC/REL without *fp* reports 32 vulnerable instructions in 1580 seconds, while BINSEC/REL with *fp* reports 2 vulnerable *basic blocks* (covering the 32 vulnerable instructions) in only 146 seconds.

Comparison vs. Standard SE (RQ5). Standard SE is directly implemented in the REL module and models a single execution of the program with exploration queries *but without insecurity queries*. We also consider a recent implementation of read-over-write [66] implemented as a formula pre-processing, posterior to SE (*PostRow*). Results are presented in Table VII.

	#I	#I/s	#Q	T	⧗
SE	440k	15.1	23453	29122	7
SE+PostRow[66]	509k	18.5	27252	27587	7
SE+FlyRow	22.8M	6804	2688	3346	0
RelSE	320k	5.4	96919	59316	14
RelSE+PostRow	254k	4.0	75043	63693	16
BINSEC/REL	22.8M	3861	3980	5895	0

Table VII: Performances of relational symbolic execution compared to standard symbolic execution with/without binary level simplifications.

- The overhead of BINSEC/REL compared to our best setting for SE (*SE+FlyRow*), in terms of speed (#I/s), is only $\times 1.8$. Hence CT comes almost for free on top of standard SE. This is consistent with the fact that our simplifications discard most insecurity queries, letting only the exploration queries which are also part of SE.
- *FlyRow* completely outperforms *PostRow*. First, *PostRow* is not designed for relational verification and must reason about pairs of memory. Second, *PostRow* simplifications are not propagated along the execution and must be recomputed for every query, producing a significant simplification-time overhead. On the contrary, *FlyRow* models a single memory containing relational values and propagates along the symbolic execution.
- *FlyRow* also improves the performance of standard SE by a factor 450, performing much better than *PostRow* in our experiments.

Conclusion (RQ3, RQ4, RQ5). BINSEC/REL performs significantly better than previous approaches to relational symbolic execution ($\times 715$ speedup vs. *RelSE*). The very main source of improvement is the *FlyRow* on-the-fly simplification ($\times 569$ speedup vs. *RelSE*, $\times 60$ less insecurity queries). Note that, in our context, *FlyRow* completely outperforms state-of-the-art binary-level simplifications, as they are not designed to efficiently cope with relational properties and introduce a significant simplification-overhead at every query. Fault-packing and untainting, while effective over *RelSE*, have a much slighter impact once *FlyRow* is activated; fault-packing can still be useful when report precision is not the main concern. Finally, in our experiments, *FlyRow* significantly improves the performance of standard SE ($\times 450$ speedup).

VIII. DISCUSSION

Implementation limitations. Our implementation shows three main limitations commonly found in research prototypes: it does not support dynamic libraries – executable must be statically linked or stubs must be provided for external function calls, it does not implement predefined syscall stubs, and it does not support floating point instructions. These problems are orthogonal to the core contribution of this paper and the two first ones are essentially engineering tasks. Moreover, the prototype is already efficient on real-world case studies.

Threats to validity in experimental evaluation. We assessed the effectiveness of our tool on several known secure and insecure real-world cryptographic binaries, many of them taken from prior studies. All results have been crosschecked with the expected output, and manually reviewed in case of deviation.

Our prototype is implemented as part of BINSEC [55], whose efficiency and robustness have been demonstrated in prior large scale studies on both adversarial code and managed code [61], [74]–[76]. The IR lifting part has been positively evaluated in external studies [53], [77] and the symbolic engine features aggressive formula optimizations [66]. All our experiments use the same search heuristics (depth-first) and, for bounded-verification, smarter heuristics do not change the performances. Also, we tried Z3 and confirmed the better performance of Boolector.

Finally, we compare our tool to our own versions of *SC* and *RelSE*, primarily because none of the existing tools can be easily adapted for our setting, and also because this allows comparing very close implementations.

IX. Related Work

Related work has already been lengthly discussed along the paper. We add here only a few additional discussions, as well as an overview of existing SE-based tools for information flow (Table VIII) and an overview of (other) existing automatic analyzers for CT (Table IX), partly taken from [16].

Tool	Target	NI	Technique	P/BV/BF/C	≈XP max	I_u/s
RelSym [49]	imp-for	✓	RelSE	✓/✓/✓/✓	10 loc	NA
IF-exploit[41]	Java	✓	SC	✓/✓/✓/✓	20 loc	NA
Type-SC-SE[42]	C	✓	type-based SC	✗/✗/✓/✓	20 loc	NA
Casym [17]	LLVM	✓	SC+over-approx	✓/✓/✗/✗	200 (C)	NA
IF-low-level [40]	**binary**	✓	SC + invariants	✓/✓/✗/✗	250 I_s	NA
IF-firmware[43]	**binary**	✗	SC + concretize	✗/✗/✓/✓	500k I_u	260
CacheD [20]	**binary**	✗	concret+tainting	✗/✗/✓/✓	31M I_u	2010
BINSEC/REL	**binary**	✗	RelSE + simpl.	✗/✓/✓/✓	10M I_u	3861

Table VIII: SE-based tools for Information Flow. NI indicates whether the technique handles general non-interference (diverging paths) or not (CT-like properties), P: proof, BV: bounded-verification, BF: bug-finding, C: counterexample. I_s: static instr., I_u: unrolled instr., NA: non-applicable.

Self-compositon and SE has first been used by Milushev *et al.* [42]. They use type-directed self-composition and dynamic symbolic execution to find bugs of *noninterference* but they do not address scalability and their experiments are limited to toy examples. The main issues here are the quadratic explosion of the search space (due to the necessity of considering diverging paths) and the complexity of the underlying formulas. Later works [40], [41] suffer from the same problems.

Instead of considering the general case of noninterference, we focus on CT, and we show that it remains tractable for SE with adequate optimizations.

Relational symbolic execution. *Shadow Symbolic Execution* [48], [78] aims at efficiently testing evolving softwares by focusing on the new behaviors introduced by a patch. The paper introduces the idea of *sharing formulas* across two executions in the same SE instance. The term *relational symbolic execution* has been coined more recently [49] but this work is limited to a simple toy imperative language and do not address scalability.

We maximize sharing between pairs of executions, as ShadowSE does, but we also develop specific optimizations tailored to the case of binary-level CT. Experiments show that our optimizations are crucial in this context.

Symbolic execution for CT. Only three previous works in this category achieve scalability, yet at the cost of either precision or soundness. Wang et al. [20] and Subramanyan *et al.* [43] sacrifice soundness for scalability (no bounded-verification). The former performs symbolic execution on fully concrete traces and only symbolize the secrets. The latter concretizes memory accesses. In both cases, they may miss feasible paths as well as vulnerabilities. Brotzman *et al.* [17]

Tool	Target	Analysis	Technique	P/BV/BF/C
ct-ai [15]	C	static	abstract-interpretation	✓/✓/✗/✗
FlowTracker [81]	LLVM	static	type-system	✓/✓/✗/✗
ct-verif [16]	LLVM	static	logical, product-programs	✓/✓/✗*/ ✗
Casym [17]‡	LLVM	static	over-approx. SE	✓/✓/✗/✗
VirtualCert† [8]	x86	static	type-system	✓/✓/✗/✗
ctgrind [18]	**binary**	dynamic	Valgrind	✗/✗/✗/✓
CacheAudit [24]‡	**binary**	static	abstract-interpretation	✓/✓/✗/✗
CacheD [20]‡	**binary**	dynamic	DSE	✗/✗/✓/✓
BINSEC/REL	**binary**	SE	RelSE + simpl.	✗/✓/✓/✓

Table IX: Automatic analysis tools for CT-like properties (see [16]). * ct-verif can be incomplete because of invariant inference. † As part of CompCert, cannot be used on arbitrary executables. ‡ Also implements a cache model.

take the opposite side and sacrifice precision for scalability (no bug-finding). Their analysis scales by over-approximating loops and resetting the symbolic state at chosen code locations.

We adopt a different approach and scale by heavy formula optimizations, allowing us to keep both correct bug-finding (BF) and correct bounded-verification (BV). Interestingly, our method is faster than these approximated ones. *We propose the first technique for CT-verification at binary-level that is correct for BF and BV and scales on real world cryptographic examples.* Moreover, our technique is compatible with the previous approximations for extra-scaling.

Other Methods for CT Analysis. *Static approaches* based on sound static analysis [8], [14]–[16], [22]–[24], [79]–[81] give formal guarantees that a program is free from time side-channels but they cannot find bugs when a program is rejected. Some works also propose program transformations to make a program secure [17], [79], [80], [82], [83] but they consider less capable attackers and target higher-level code. *Dynamic approaches* for constant-time are precise (they find real violations) but limited to a subset of the execution traces, hence they are not complete. These techniques include statistical analysis [84], dynamic binary instrumentation [18], [21], and dynamic symbolic execution (DSE) [19].

X. Conclusion

We tackle the problem of designing an automatic and efficient binary-level analyzer for *constant-time*, enabling both bug-finding and bounded-verification on real-world cryptographic implementations. Our approach is based on *relational symbolic execution* together with original *dedicated optimizations* reducing the overhead of relational reasoning and allowing for a significant speedup. Our prototype, BINSEC/REL, is shown to be highly efficient compared to alternative approaches. We used it to perform extensive binary-level CT analysis for a wide range of cryptographic implementations and to automate and extend a previous study of CT preservation by compilers. We found three vulnerabilities that slipped through previous manual and automated analyses, and we discovered that `gcc -O0` and backend passes of `clang` introduce violations of CT out of reach of state-of-the-art CT verification tools at LLVM or source level.

References

[1] D. Brumley and D. Boneh, "Remote timing attacks are practical", *Computer Networks*, vol. 48, no. 5, 2005.

[2] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical", in *ESORICS*, 2011.

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

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

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES", in *CT-RSA*, 2006.

[6] C. Percival, "Cache missing for fun and profit", 2009.

[7] E. Ronen, K. G. Paterson, and A. Shamir, "Pseudo constant time implementations of TLS are only pseudo secure", in *CCS*, 2018.

[8] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography", in *CCS*, 2014.

[9] T. Pornin, *BearSSL*. [Online]. Available: https://www.bearssl.org/ (visited on 05/23/2019).

[10] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library", in *LATINCRYPT*, 2012.

[11] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library", in *CCS*, 2017.

[12] 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.

[13] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas, "When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015", in *CANS*, 2016.

[14] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, "Formal verification of side-channel countermeasures using self-composition", *Sci. Comput. Program.*, vol. 78, no. 7, 2013.

[15] S. Blazy, D. Pichardie, and A. Trieu, "Verifying constant-time implementations by abstract interpretation", in *ESORICS*, 2017.

[16] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying Constant-Time Implementations.", in *USENIX*, 2016.

[17] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "CaSym: Cache aware symbolic execution for side channel detection and mitigation", in *S&P*, 2019.

[18] A. Langley, *ImperialViolet - Checking that functions are constant time with Valgrind*, 2010. [Online]. Available: https://www.imperialviolet.org/2010/04/01/ctgrind.html.

[19] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller, "Quantifying the information leak in cache attacks via symbolic execution", in *MEMOCODE*, 2017.

[20] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software", in *USENIX*, 2017.

[21] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries", in *ACSAC*, 2018.

[22] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic Quantification of Cache Side-Channels", in *CAV*, 2012.

[23] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A Tool for the Static Analysis of Cache Side Channels", *ACM Transactions on Information and System Security*, vol. 18, no. 1, 2015.

[24] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks", in *PLDI*, 2017.

[25] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub, "Jasmin: High-assurance and high-speed cryptography", in *CCS*, 2017.

[26] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. T. V. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code", in *USENIX*, 2017.

[27] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "Fact: A flexible, constant-time programming language", in *SecDev*, 2017.

[28] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches", in *CCS*, 2016.

[29] F. Liu, Q. Ge, Y. Yarom, F. McKeen, C. V. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing", in *HPCA*, 2016.

[30] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: The missing OS abstraction", in *EuroSys*, 2019.

[31] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory", in *USENIX*, 2017.

[32] M. R. Clarkson and F. B. Schneider, "Hyperproperties", *Journal of Computer Security*, vol. 18, no. 6, 2010.

[33] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition", in *CSFW*, 2004.

[34] T. Terauchi and A. Aiken, "Secure information flow as a safety problem", in *SAS*, 2005.

[35] G. Balakrishnan and T. W. Reps, "WYSINWYX: what you see is not what you execute", *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, 2010.

[36] A. Djoudi, S. Bardin, and É. Goubault, "Recovering high-level conditions from binary programs", in *FM*, 2016.

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

[38] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later”, *Communications of the ACM*, vol. 56, no. 2, 2013.
[39] E. Bounimova, P. Godefroid, and D. A. Molnar, “Billions and billions of constraints: Whitebox fuzz testing in production”, in *ICSE*, 2013.
[40] M. Balliu, M. Dam, and R. Guanciale, “Automating information flow analysis of low level code”, in *CCS*, 2014.
[41] Q. H. Do, R. Bubel, and R. Hähnle, “Exploit generation for information flow leaks in object-oriented programs”, in *SEC*, 2015.
[42] D. Milushev, W. Beck, and D. Clarke, “Noninterference via symbolic execution”, in *Formal Techniques for Distributed Systems*, 2012.
[43] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. M. Fung, “Verifying information flow properties of firmware using symbolic execution”, in *DATE*, 2016.
[44] N. Benton, “Simple relational correctness proofs for static analyses and program transformations”, in *POPL*, 2004.
[45] G. Barthe, J. M. Crespo, and C. Kunz, “Relational verification using product programs”, in *FM*, 2011.
[46] T. H. Austin and C. Flanagan, “Multiple facets for dynamic information flow”, in *POPL*, 2012.
[47] M. Ngo, N. Bielova, C. Flanagan, T. Rezk, A. Russo, and T. Schmitz, “A better facet of dynamic information flow control”, in *WWW (Companion Volume)*, 2018.
[48] H. Palikareva, T. Kuchta, and C. Cadar, “Shadow of a doubt: Testing for divergences between software versions”, in *ICSE*, 2016.
[49] G. P. Farina, S. Chong, and M. Gaboardi, “Relational symbolic execution”, in *PPDP*, 2019.
[50] N. J. AlFardan and K. G. Paterson, “Lucky thirteen: Breaking the TLS and DTLS record protocols”, in *S&P*, 2013.
[51] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.”, in *OSDI*, 2008.
[52] T. Avgerinos, D. Brumley, J. Davis, R. Goulden, T. Nighswander, A. Rebert, and N. Williamson, “The mayhem cyber reasoning system”, *IEEE Security & Privacy*, vol. 16, no. 2, 2018.
[53] V. Chipounov, V. Kuznetsov, and G. Candea, “The S2E platform: Design, implementation, and applications”, *ACM Trans. Comput. Syst.*, vol. 30, no. 1, 2012.
[54] 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.
[55] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion, “BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis”, in *SANER*, 2016.
[56] J. C. King, “Symbolic execution and program testing”, *Commun. ACM*, vol. 19, no. 7, 1976.
[57] J. Vanegue and S. Heelan, “SMT solvers in software security”, in *WOOT*, 2012.
[58] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, “AEG: automatic exploit generation”, in *NDSS*, 2011.
[59] E. J. Schwartz, T. Avgerinos, and D. Brumley, “Q: exploit hardening made easy”, in *USENIX*, 2011.
[60] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, “A generic approach to automatic deobfuscation of executable code”, in *S&P*, 2015.
[61] S. Bardin, R. David, and J. Marion, “Backward-bounded DSE: targeting infeasibility questions on obfuscated codes”, in *S&P*, 2017.
[62] J. Salwan, S. Bardin, and M. Potet, “Symbolic deobfuscation: From virtualized code back to the original”, in *DIMVA*, 2018.
[63] C. Barrett, P. Fontaine, and C. Tinelli, “The SMT-LIB Standard: Version 2.6”, Department of Computer Science, The University of Iowa, Tech. Rep., 2017.
[64] *FixedSizeBitVectors Theory, SMT-LIB*. [Online]. Available: http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml (visited on 04/02/2019).
[65] *ArraysEx Theory, SMT-LIB*. [Online]. Available: http://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml (visited on 04/02/2019).
[66] B. Farinier, R. David, S. Bardin, and M. Lemerre, “Arrays made simpler: An efficient, scalable and thorough preprocessing”, in *LPAR*, 2018.
[67] D. J. Bernstein, “Curve25519: New diffie-hellman speed records”, in *Public Key Cryptography*, 2006.
[68] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, “The BINCOA framework for binary code analysis”, in *CAV*, 2011.
[69] G. Barthe, B. Grégoire, and V. Laporte, “Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time"”, in *CSF*, 2018.
[70] A. Niemetz, M. Preiner, and A. Biere, “Boolector 2.0 system description”, *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, 2014.
[71] *SMT-COMP*. [Online]. Available: https://smt-comp.github.io/2019/results.html (visited on 10/11/2019).
[72] *Imdea-software/verifying-constant-time*. [Online]. Available: https://github.com/imdea-software/verifying-constant-time (visited on 10/13/2019).
[73] D. J. Wheeler and R. M. Needham, “Tea, a tiny encryption algorithm”, in *FSE*, 1994.
[74] F. Recoules, S. Bardin, R. B. Bonichon, L. Mounier, and M.-L. Potet, “Get rid of inline assembly through verification-oriented lifting”, in *ASE*, 2019.
[75] B. Farinier, S. Bardin, R. Bonichon, and M. Potet, “Model generation for quantified formulas: A taint-based approach”, in *CAV (2)*, 2018.
[76] R. David, S. Bardin, J. Feist, L. Mounier, M. Potet, T. D. Ta, and J. Marion, “Specification of concretization

and symbolization policies in symbolic execution", in *ISSTA*, 2016.

[77] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha, "B2r2: Building an efficient front-end for binary analysis", in *The BAR Workshop*, Internet Society, 2019.

[78] C. Cadar and H. Palikareva, "Shadow symbolic execution for better testing of evolving software", in *ICSE*, 2014.

[79] J. Agat, "Transforming out timing leaks", in *POPL*, 2000.

[80] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks", in *ICISC*, 2005.

[81] B. Rodrigues, F. M. Q. Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection", in *CC*, 2016.

[82] S. Chattopadhyay and A. Roychoudhury, "Symbolic verification of cache side-channel freedom", *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 37, no. 11, 2018.

[83] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair", in *ISSTA*, 2018.

[84] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?", in *DATE*, 2017.

APPENDIX

A. Full Set of rules

1) Concrete Evaluation: The full set of rules for the concrete evaluation is reported in Figure 7.

2) Symbolic Evaluation: The full set of rules for the symbolic evaluation is reported in Figure 8.

Expr

$$\text{CST}\ \frac{}{(l, r, m)\ \texttt{bv} \vdash \texttt{bv}}$$

$$\text{VAR}\ \frac{}{(l, r, m)\ \texttt{v} \vdash r\ \texttt{v}} \qquad \text{UNOP}\ \frac{(l, r, m)\ e \vdash \texttt{bv}}{(l, r, m)\ \blacklozenge_u e \vdash \blacklozenge_u \texttt{bv}}$$

$$\text{BINOP}\ \frac{(l, r, m)\ e_1 \vdash \texttt{bv}_1 \qquad (l, r, m)\ e_2 \vdash \texttt{bv}_2}{(l, r, m)\ e_1 \blacklozenge_b e_2 \vdash \texttt{bv}_1 \diamond_b \texttt{bv}_2}$$

$$\text{LOAD}\ \frac{(l, r, m)\ e \vdash_t \texttt{bv}}{(l, r, m)\ \texttt{@}e \vdash_{t \cdot [\texttt{bv}]} m\ \texttt{bv}}$$

Instr

$$\text{S\_JUMP}\ \frac{P.l = \texttt{goto}\ l'}{(l, r, m) \xrightarrow[{[l]}]{} (l', r, m)}$$

$$\text{D\_JUMP}\ \frac{P.l = \texttt{goto}\ e \qquad (l, r, m)\ e \vdash_t \texttt{bv} \qquad l' \triangleq \mathit{to\_loc}(\texttt{bv})}{(l, r, m) \xrightarrow[t \cdot [l']]{} (l', r, m)}$$

$$\text{ITE-TRUE}\ \frac{P.l = \texttt{ite}\ e\ ?\ l_1 : l_2 \qquad (l, r, m)\ e \vdash_t \texttt{bv} \qquad \texttt{bv} \neq 0}{(l, r, m) \xrightarrow[t \cdot [l_1]]{} (l_1, r, m)}$$

$$\text{ITE-FALSE}\ \frac{P.l = \texttt{ite}\ e\ ?\ l_1 : l_2 \qquad (l, r, m)\ e \vdash_t \texttt{bv} \qquad \texttt{bv} = 0}{(l, r, m) \xrightarrow[t \cdot [l_2]]{} (l_2, r, m)}$$

$$\text{ASSIGN}\ \frac{P.l = \texttt{v}\ := \ e \qquad (l, r, m) e \vdash_t \texttt{bv}}{(l, r, m) \xrightarrow[t]{} (l + 1, r[\texttt{v} \mapsto \texttt{bv}], m)}$$

$$\text{STORE}\ \frac{P.l = \texttt{@}e := e' \qquad (l, r, m)\ e \vdash_t \texttt{bv} \qquad (l, r, m)\ e' \vdash_{t'} \texttt{bv}'}{(l, r, m) \xrightarrow[t' \cdot t \cdot [\texttt{bv}]]{} (l + 1, r, m[\texttt{bv} \mapsto \texttt{bv}'])}$$

Figure 7: Concrete evaluation of DBA instructions and expressions, where $\cdot$ is the concatenation of leakages and $\mathit{to\_loc} : \mathit{BV}_{32} \to \mathit{Loc}$ converts a bitvector to a location.

B. Proofs

We recall essential elements that we will use in the proofs.

Proposition 1. *Concrete semantics is deterministic, c.f. rules of the concrete semantics in Fig. 7.*

Proposition 2. *If a program P is constant-time up to k then then for all $j \leq k$, P is constant-time up to j.*

Hypothesis 1. *Symbolic execution does not get stuck unless secLeak evaluates to false. In particular, this implies that P is defined on all locations computed during the symbolic execution.*

1) Proof of Relative Completeness of RelSE (Theorem 1):

Theorem 1 (Relative Completeness of RelSE)**.** *Let P be a program constant-time up to k and s_0 be a symbolic initial configuration for P. For every concrete states c_0, c_k, c'_0, c'_k, and model M such that $c_0 \approx_l^M s_0 \ \wedge\ c'_0 \approx_r^M s_0$, if $c_0 \xrightarrow[t]{}^k c_k$ and $c'_0 \xrightarrow[t']{}^k c'_k$ with $t = t'$ then there exists a symbolic configuration s_k and a model M' such that:*

$$s_0 \rightsquigarrow^k s_k \ \wedge\ c_k \approx_l^{M'} s_k \ \wedge\ c'_k \approx_r^{M'} s_k$$

Proof. (Induction on k). Case $k = 0$ is trivial.

Let c_k and c'_k be concrete configurations and s_k a symbolic configuration for which the inductive hypothesis holds up to $k - 1$. We need to show that Theorem 1 still holds at step k, meaning that if P is constant-time up to step k, then for each concrete steps $c_{k-1} \xrightarrow[t]{} c_k$ and $c'_{k-1} \xrightarrow[t']{} c'_k$ such that $t = t'$, we need to show that we can perform a step in the symbolic execution $s_{k-1} \rightsquigarrow s_k$ and that $c_k \approx_l^M s_k \wedge c'_k \approx_r^M s_k$ holds.

We can proceed case by case on the concrete evaluation of c_{k-1} and c'_{k-1}.

Case STORE: In the concrete execution, the instruction $\texttt{@}e_{idx} := e_{val}$ is evaluated and leaks the index e_{idx} of the store. By Proposition 1, concrete semantics of the STORE rule, and because $t = t'$, we have:

$$c_{k-1}\ e_{idx} \vdash \texttt{bv}_\texttt{i} \text{ and } c'_{k-1}\ e_{idx} \vdash \texttt{bv}_\texttt{i} \tag{1}$$

■ First, we show that there exists a step from $s_{k-1} \triangleq \left(l, \rho, \widehat{\mu}, \pi\right)$ in the symbolic execution. We have to consider the case where the symbolic evaluation of expressions is not stuck and the case where the evaluation of an expression gets

Expr

CST $\dfrac{}{(\rho,\widehat{\mu})\ \mathtt{bv} \vdash \langle bv\rangle}$ VAR $\dfrac{}{(\rho,\widehat{\mu})\ \mathtt{v} \vdash \rho\ \mathtt{v}}$

$$\text{UNOP}\ \dfrac{(\rho,\widehat{\mu})\ e \vdash \widehat{\phi} \qquad \widehat{\varphi} \triangleq \diamond_u\, \widehat{\phi}}{(\rho,\widehat{\mu})\ \blacklozenge_u e \vdash \widehat{\varphi}}$$

$$\text{BINOP}\ \dfrac{(\rho,\widehat{\mu})\ e_1 \vdash \widehat{\phi} \qquad (\rho,\widehat{\mu})\ e_2 \vdash \widehat{\psi} \qquad \widehat{\varphi} \triangleq \widehat{\phi} \diamond_b \widehat{\psi}}{(\rho,\widehat{\mu})\ e_1 \blacklozenge_b e_2 \vdash \widehat{\varphi}}$$

$$\text{LOAD}\ \dfrac{\begin{array}{c}(\rho,\widehat{\mu})\ e \vdash \widehat{\phi}\\ \widehat{\varphi} \triangleq \langle select(\widehat{\mu}_{|l},\widehat{\phi}_{|l}) \mid select(\widehat{\mu}_{|r},\widehat{\phi}_{|r})\rangle \qquad secLeak\,(\widehat{\phi})\end{array}}{(\rho,\widehat{\mu})\ @e \vdash \widehat{\varphi}}$$

Instr

$$\text{S\_JUMP}\ \dfrac{P.l = \mathtt{goto}\ l'}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l',\rho,\widehat{\mu},\pi)}$$

$$\text{D\_JUMP}\ \dfrac{\begin{array}{c}P.l = \mathtt{goto}\ e \qquad (\rho,\widehat{\mu})\ e \vdash \widehat{\varphi} \qquad \pi' \triangleq \pi \wedge (\widehat{\varphi}_{|l} = \widehat{\varphi}_{|r})\\ M \vDash \pi' \qquad l' \triangleq M(\widehat{\varphi}_{|l}) \qquad secLeak\,(\widehat{\varphi})\end{array}}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l',\rho,\widehat{\mu},\pi')}$$

ITE-TRUE
$$\dfrac{\begin{array}{c}P.l = \mathtt{ite}\ e\ ?\ l_{true} : l_{false} \qquad l' \triangleq l_{true} \qquad (\rho,\widehat{\mu})\ e \vdash \widehat{\varphi}\\ \pi' \triangleq \pi \wedge (true = \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r}) \qquad \vDash \pi' \qquad secLeak\,(\widehat{\varphi})\end{array}}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l',\rho,\widehat{\mu},\pi')}$$

ITE-FALSE
$$\dfrac{\begin{array}{c}P.l = \mathtt{ite}\ e\ ?\ l_{true} : l_{false} \qquad l' \triangleq l_{false} \qquad (\rho,\widehat{\mu})\ e \vdash \widehat{\varphi}\\ \pi' \triangleq \pi \wedge (false = \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r}) \qquad \vDash \pi' \qquad secLeak\,(\widehat{\varphi})\end{array}}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l',\rho,\widehat{\mu},\pi')}$$

$$\text{ASSIGN}\ \dfrac{\begin{array}{c}P.l = v := e \qquad (\rho,\widehat{\mu})\ e \vdash \widehat{\varphi} \qquad \widehat{\varphi}' \triangleq canonical\,(\widehat{\varphi})\\ \rho' \triangleq \rho[v \mapsto \widehat{\varphi}'] \qquad \pi' \triangleq \pi \wedge (\widehat{\varphi}'_{|l} = \widehat{\varphi}_{|l}) \wedge (\widehat{\varphi}'_{|r} = \widehat{\varphi}_{|r})\end{array}}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l+1,\rho',\widehat{\mu},\pi')}$$

STORE
$$\dfrac{\begin{array}{c}P.l = @e := e' \qquad l' = l+1 \qquad (\rho,\widehat{\mu})\ e \vdash \widehat{\varphi}\\ (\rho,\widehat{\mu})\ e' \vdash \widehat{\phi} \qquad \widehat{\mu}' \triangleq \langle store(\widehat{\mu}_{|l},\widehat{\varphi}_{|l},\widehat{\phi}_{|l}) \mid store(\widehat{\mu}_{|r},\widehat{\varphi}_{|r},\widehat{\phi}_{|r})\rangle\\ \pi' \triangleq \pi \wedge \widehat{\mu}'_{|l} = store(\widehat{\mu}_{|l},\widehat{\varphi}_{|l},\widehat{\phi}_{|l}) \wedge \widehat{\mu}'_{|r} = store(\widehat{\mu}_{|r},\widehat{\varphi}_{|r},\widehat{\phi}_{|r})\\ secLeak\,(\widehat{\varphi})\end{array}}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l',\rho,\widehat{\mu}',\pi')}$$

Figure 8: Symbolic evaluation of DBA instructions and expressions where $canonical\,(\widehat{\varphi})$ returns $\widehat{\varphi}$ if it is in canonical form or a temporary variable otherwise; and $\diamond_u$ (resp. $\diamond_b$) is the logical counterpart of the concrete operator $\blacklozenge_u$ (resp. $\blacklozenge_b$).

stuck.

Case evaluation of expressions is not stuck. Let ι be the symbolic index such that $\big(\rho,\widehat{\mu}\big)\ e_{idx} \vdash \widehat{\iota}$. To apply the rule STORE, we must ensure that $secLeak\,(\widehat{\iota})$ holds. If $\widehat{\iota} = \langle\iota\rangle$ then $secLeak$ is true, otherwise we must show $\nvDash \pi \wedge \widehat{\iota}_{|l} \neq \widehat{\iota}_{|r}$. We show that by contradiction.

Assume that there exists M' such that $M' \vDash \pi \wedge \widehat{\iota}_{|l} \neq \widehat{\iota}_{|r}$ and let $M'(\widehat{\iota}_{|l}) = \mathtt{bv_i}$ and $M'(\widehat{\iota}_{|r}) = \mathtt{bv'_i}$. Notice that $\mathtt{bv_i} \neq \mathtt{bv'_i}$.

Let $d_0, d'_0, d_{k-1}, d'_{k-1}$ be concrete configurations such that $d_0 \approx_l^{M'} s_0$, $d'_0 \approx_r^{M'} s_0$, $d_{k-1} \approx_l^{M'} s_{k-1}$, and $d'_{k-1} \approx_r^{M'} s_{k-1}$. From Theorem 2, there are concrete executions $d_0 \underset{t}{\longrightarrow}^{k-1} d_{k-1}$ and $d'_0 \underset{t'}{\longrightarrow}^{k-1} d'_{k-1}$. Because $d_k \approx_l^{M'} s_{k-1}$ and $d'_{k-1} \approx_r^{M'} s_{k-1}$, we know that d_{k-1} and d'_{k-1} also evaluate the instruction $@e_{idx} := e_{val}$, and that

$$d_{k-1}\ e_{idx} \vdash M'(\widehat{\iota}_{|l}) \text{ and } d'_{k-1}\ e_{idx} \vdash M'(\widehat{\iota}_{|r})$$

Therefore, we have $d_0 \xrightarrow[t\cdot[\mathtt{bv_i}]]{}^{k} d_k$ and $d'_0 \xrightarrow[t'\cdot[\mathtt{bv'_i}]]{}^{k} d'_k$ with $\mathtt{bv} \neq \mathtt{bv'_i}$, meaning that P is not constant-time at step k. This contradicts the hypothesis that P is constant-time up to k, hence $\nvDash \pi \wedge \widehat{\iota}_{|l} \neq \widehat{\iota}_{|r}$ and $secLeak$ holds. Therefore, there exists a step $s_{k-1} \rightsquigarrow s_k$ in the symbolic execution.

Case $\big(\rho,\widehat{\mu}\big)\ e_{idx}$ is stuck. From Hypothesis 1, $secLeak$ evaluates to false in the evaluation of the expression. With the same reasoning as in the previous case, this implies that P is not constant-time at step k, which contradicts the hypothesis of Theorem 1. Hence the symbolic evaluation of e_{idx} cannot be stuck.

Case $\big(\rho,\widehat{\mu}\big)\ e_{val}$ is stuck is analogous.

■ Now, we need to show that there is a model M' such that $c_k \approx_l^{M'} s_k\ \wedge\ c'_k \approx_r^{M'} s_k$ holds.

Recall that the program location in s_{k-1} evaluates to a store instruction $@e_{idx} := e_{val}$ and let $\widehat{\iota}$ be the symbolic index and $\widehat{\nu}$ be the symbolic value, meaning that $\big(\rho,\widehat{\mu}\big)\ e_{idx} \vdash \widehat{\iota}$ and $\big(\rho,\widehat{\mu}\big)\ e_{val} \vdash \widehat{\nu}$. Let $s_k \triangleq \big(l_k, \rho_k, \widehat{\mu}_k, \pi_k\big)$.

First we need to show that the location in configurations c_k, c'_k and s_k are identical. Because concrete and symbolic STORE rules increment the program location by 1 and because the program locations are identical in c_{k-1}, c'_{k-1} and s_{k-1} (from induction hypothesis), the program locations are still identical in c_k, c'_k and s_k.

Second, we have to show that there exists M' such that $M' \vDash \pi_k$ and that for all expression e, either the symbolic evaluation gets stuck on e, or $\big(\rho_k,\widehat{\mu}_k\big)\ e \vdash \widehat{\varphi}$ and

$$\begin{aligned} M'(\widehat{\varphi}_{|l}) = \mathtt{bv} &\iff c_k\ e \vdash \mathtt{bv}\ \wedge \\ M'(\widehat{\varphi}_{|r}) = \mathtt{bv} &\iff c'_k\ e \vdash \mathtt{bv} \end{aligned} \tag{2}$$

We can build the new model M' as

$$\begin{aligned} M' \triangleq M[\widehat{\mu}_k \mapsto \langle m_l \mid m_r\rangle] \\ \text{where } m_l \triangleq M(\widehat{\mu}_{|l})[M(\widehat{\iota}_{|l}) \mapsto M(\widehat{\nu}_{|l})] \\ \text{and } m_r \triangleq M(\widehat{\mu}_{|r})[M(\widehat{\iota}_{|r}) \mapsto M(\widehat{\nu}_{|r})] \end{aligned}$$

Intuitively, M' is equal to the old model M in which we add the new symbolic memory $\widehat{\mu}_k$, mapping to the concrete value of the old memory $M(\widehat{\mu})$ where the index $M(\widehat{\iota})$ maps to the value $M(\widehat{\nu})$. Notice that $M' \vDash \pi_k$.

Case of the left projection (right case is analogous). We can prove by induction on the structure of expressions that for any expression, if s_k does not get stuck then Eq. (2) holds for the model M'. Note that only the memory is updated from step $k-1$ to step k, meaning that c_k, s_k, and M' only differ from c_{k-1}, s_{k-1} and M on expressions involving the memory. Thus, we only need to consider the rule LOAD, as the proof for other rules directly follows from the induction hypothesis and the definition of M'.

Assume an expression $@e$ such that s_k does not get stuck and let $\big(\rho_k,\widehat{\mu}_k\big)\ e \vdash \widehat{\iota'}$ end $\big(\rho_k,\widehat{\mu}_k\big)\ @e \vdash \widehat{\nu'}$. We show that if Eq. (2) holds for the expression e, then it holds for the expression $@e$. Formally, we must show that if $M'(\widehat{\iota'}_{|l}) = \texttt{bv}_{\texttt{idx}} \iff c_k\ e \vdash \texttt{bv}_{\texttt{idx}}$ then $M'(\widehat{\nu'}_{|l}) = \texttt{bv}_{\texttt{val}} \iff c_k\ @e \vdash \texttt{bv}_{\texttt{val}}$.

First, we can simplify $M'(\widehat{\nu'}_{|l})$ as

$$
\begin{aligned}
M'(\widehat{\nu'}_{|l}) &= M'(select(\widehat{\mu}_{k|l}, \widehat{\iota'}_{|l})) \text{ by symbolic rule LOAD} \\
&= M(\widehat{\mu}_{|l})[M(\hat{\iota}_{|l}) \mapsto M(\widehat{\nu}_{|l})][M'(\widehat{\iota'}_{|l})] \text{ by def. of } M'
\end{aligned}
$$

From this point, there are two cases:

- The address of the load is the same as the address of the previous store: $M'(\widehat{\iota'}_{|l}) = M(\hat{\iota}_{|l})$, therefore $M'(\widehat{\nu'}_{|l}) = M(\widehat{\nu}_{|l})$.
  From the induction hypothesis, the concrete index of the load evaluates to $M'(\widehat{\iota'}_{|l})$, i.e. $c_k\ e \vdash M'(\widehat{\iota'}_{|l})$ which can be rewritten as $c_k\ e \vdash M(\hat{\iota}_{|l})$. From concrete rule STORE and $c_{k-1} \approx_l^M s_{k-1}$, we know that the concrete memory from c_{k-1} to c_k is updated at index $M(\hat{\iota}_{|l})$ to map to the value $M(\widehat{\nu}_{|l})$, which leads to $c_k @e \vdash M(\widehat{\nu}_{|l})$ and, by rewriting, to $c_k\ @e \vdash M'(\widehat{\nu'}_{|l})$. Therefore we have shown that $M'(\widehat{\nu'}_{|l}) = \texttt{bv}_{\texttt{val}} \iff c_k\ @e \vdash \texttt{bv}_{\texttt{val}}$
- The address of the load is different from the address of the previous store: $M'(\widehat{\iota'}_{|l}) \neq M(\hat{\iota}_{|l})$, therefore $M'(\widehat{\nu'}_{|l}) = M(\widehat{\mu}_{|l})[M'(\widehat{\iota'}_{|l})]$.
  From the induction hypothesis, the concrete index of the load evaluates to $M'(\widehat{\iota'}_{|l})$, i.e. $c_k\ e \vdash M'(\widehat{\iota'}_{|l})$. From concrete rule STORE, we know that the concrete memory from c_{k-1} to c_k is only updated at address $M(\hat{\iota}_{|l})$ and untouched at address $M'(\widehat{\iota'}_{|l})$. Plus, we know from $c_{k-1} \approx_l^M s_{k-1}$ that address $M'(\widehat{\iota'}_{|l})$ maps to $M(\widehat{\mu}_{|l})[M'(\widehat{\iota'}_{|l})]$ in c_{k-1}. Therefore, in configuration c_k, index $M'(\widehat{\iota'}_{|l})$ maps to $M(\widehat{\mu}_{|l})[M'(\widehat{\iota'}_{|l})]$ which, by rewriting, leads to $c_k\ @e \vdash M'(\widehat{\nu'}_{|l})$. Therefore we have shown that $M'(\widehat{\nu'}_{|l}) = \texttt{bv}_{\texttt{val}} \iff c_k\ @e \vdash \texttt{bv}_{\texttt{val}}$.

Other cases: The reasoning is analogous. For the non-deterministic rules ITE-TRUE, ITE-FALSE, and D-JUMP, because the leakages t and t' determine the control flow of the program, there exist a *unique symbolic rule* that can be applied to match the execution of both c_{k-1} and c'_{k-1}. □

2) Proof of Correctness of RelSE (Theorem 2):

Theorem 2 (Correctness of RelSE)**.** *For every symbolic configurations s_0, s_k such that $s_0 \rightsquigarrow^k s_k$ and for every concrete configurations c_0, c_k and model M, such that $c_0 \approx_p^M s_0$ and $c_k \approx_p^M s_k$, there exists a concrete execution $c_0 \to^k c_k$.*

Proof. (Induction on k). Case $k = 0$ is trivial.

Let us consider a symbolic configuration $s_{k-1} \triangleq \left(l, \rho, \widehat{\mu}, \pi\right)$ for which the induction hypothesis holds. Formally, for each model M and configurations c_0, c_{k-1} such that $c_0 \approx_p^M s_0$ and $c_{k-1} \approx_p^M s_{k-1}$, we have $c_0 \to^{k-1} c_{k-1}$. We need to show for each symbolic step $s_{k-1} \rightsquigarrow s_k$, that for each model M' and configurations c_0 and c_k such that $c_0 \approx_p^{M'} s_0$ and $c_k \approx_p^{M'} s_k$, we have $c_0 \to^k c_k$.

Let π' be the new path predicate in configuration s_k. Note that because π is a sub-formula of π', we also have $M' \vDash \pi$. Therefore, there exists c_{k-1} such that $c_{k-1} \approx_p^{M'} s_{k-1}$, and because $\approx_p^{M'}$ is a tight relation, c_{k-1} is unique. Additionally, from the induction hypothesis for all concrete configuration c_0 such that $c_0 \approx_p^{M'} s_0$, we have $c_0 \to^{k-1} c_{k-1}$.

Finally, because the symbolic execution is updated *without over-approximation*, we also have $c_{k-1} \to c_k$. Therefore, for each model M' and configuration c_0 and c_k such that $c_0 \approx_p^{M'} s_0$ and $c_k \approx_p^{M'} s_k$, we have $c_0 \to^k c_k$. □

3) Proof of CT Security (Theorem 3):

Theorem 3 (Bounded-Verification for CT)**.** *Let s_0 be a symbolic initial configuration for a program P. If the symbolic evaluation does not get stuck, then P is constant-time w.r.t. s_0. Formally, if for all k, $s_0 \rightsquigarrow^k s_k$ then for all initial configurations c_0 and c'_0 and model M such that $c_0 \approx_l^M s_0$, and $c'_0 \approx_r^M s_0$,*

$$
c_0 \simeq_L c'_0 \quad \wedge \quad c_0 \xrightarrow[t]{}^k c_k \quad \wedge \quad c'_0 \xrightarrow[t']{}^k c'_k \quad \Longrightarrow \quad t = t'
$$

Additionally, if s_0 is fully symbolic, then P is constant-time.

Proof. (Induction on k). Let s_0 be an initial symbolic configuration for which the symbolic evaluation never gets stuck. Let us consider a model M and concrete configurations $c_0 \approx_l^M s_0$, $c'_0 \approx_r^M s_0$, for which the induction hypothesis holds at step k, meaning that for all $c_k \triangleq (l, r, m)$ and $c_{k'} \triangleq (l', r', m')$ such that $c_0 \xrightarrow[t]{}^k c_k$, $c'_0 \xrightarrow[t']{}^k c'_k$, then $t = t'$. We show that Theorem 3 still holds at step $k+1$.

From Theorem 1, there exists $s_k \triangleq \left(l_s, \rho, \widehat{\mu}, \pi\right)$ such that:

$$
s_0 \rightsquigarrow^k s_k \ \wedge \ c_k \approx_l^M s_k \ \wedge \ c'_k \approx_r^M s_k \tag{3}
$$

Note that from Eq. (3) and Definition 2, we have $l_s = l = l'$, therefore the same instructions and expression are evaluated in configurations c_k, c'_k, and s_k.

Because the symbolic execution does not get stuck, there exists s_{k+1} such that $s_k \rightsquigarrow s_{k+1}$. We show by contradiction that the leakages $\texttt{bv}$ and $\texttt{bv}'$ produced by $c_k \xrightarrow[\texttt{bv}]{} c_{k+1}$ and $c'_k \xrightarrow[\texttt{bv}']{} c'_{k+1}$ are necessarily equal.

Suppose that c_k and c'_k produce distinct leakages. This can happen during the evaluation of a rule LOAD, D_JUMP, ITE, STORE.

Case LOAD: The evaluation of the expression $@e$ in configurations c_k and c'_k produces leakages $\texttt{bv}$ and $\texttt{bv}'$ and, assuming the load is insecure, we have $\texttt{bv} \neq \texttt{bv}'$.

Let $\widehat{\varphi}$ be the evaluation of the leakage in the symbolic configuration: $\left(\rho, \widehat{\mu}, e\right) \vdash \widehat{\varphi}$. From Eq. (3), Definition 2 and because symbolic execution does not get stuck, there exists M s.t. $M \vDash \pi$, $\texttt{bv} = M(\widehat{\varphi}_{|l})$ and $\texttt{bv}' = M(\widehat{\varphi}_{|r})$.

Because we assumed $\texttt{bv} \neq \texttt{bv}'$, then $M(\widehat{\varphi}_{|l}) \neq M(\widehat{\varphi}_{|r})$. Therefore, we have $M \vDash \pi \wedge \widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r}$, meaning that $secLeak$ evaluates to false and the symbolic execution is stuck. However, because s_k is non-blocking we have a contradiction. Therefore $\texttt{bv} = \texttt{bv}'$.

Cases D_JUMP**,** ITE**,** STORE: The reasoning is analogous.

We have shown that the hypothesis holds for $k+1$. If $s_0 \rightsquigarrow^{k+1} s_{k+1}$, then for all model M and low-equivalent initial configurations $c_0 \approx_l^M s_0$ and $c'_0 \approx_r^M s_0$ such that $c_0 \underset{t}{\rightarrow}^k c_k \underset{\texttt{bv}}{\longrightarrow} c_{k+1}$ and $c'_0 \underset{t'}{\rightarrow}^k c'_k \underset{\texttt{bv}'}{\longrightarrow} c'_{k+1}$ where $t = t'$, then $t \cdot [\texttt{bv}] = t' \cdot [\texttt{bv}']$. □

4) Proof of Bug-Finding for CT (Theorem 4):

Theorem 4 (Bug-Finding for CT)**.** *Let s_0 be an initial symbolic configuration for a program P. If the symbolic evaluation gets stuck in a configuration s_k then P is not constant-time. Formally, if there exists k st. $s_0 \rightsquigarrow^k s_k$ and s_k is stuck, then there exists a model M and concrete configurations $c_0 \approx_l^M s_0$, $c'_0 \approx_r^M s_0$, $c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$ such that,*

$$c_0 \simeq_L c'_0 \ \wedge \ c_0 \underset{t}{\rightarrow}^k c_k \ \wedge \ c'_0 \underset{t'}{\rightarrow}^k c'_k \wedge t \neq t'$$

Proof. Let us consider symbolic configurations s_0 and s_k such that $s_0 \rightsquigarrow^k s_k$ and s_k is stuck. This can happen during the evaluation of a rule LOAD, D_JUMP, ITE, STORE.

Case LOAD: where an expression $@e$ in the configuration $s_k \triangleq (l_s, \rho, \widehat{\mu}, \pi)$ produces a leakage $\widehat{\varphi}$ st. $(\rho, \widehat{\mu}, e) \vdash \widehat{\varphi}$.

This evaluation blocks iff $\neg secLeak(\widehat{\varphi})$, meaning that there exists a model M such that

$$M \vDash \pi \wedge (\widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r}) \tag{4}$$

Let us consider the concrete configurations c_0, c'_0, $c_k \triangleq (l, r, m)$, and $c_{k'} \triangleq (l', r', m')$ such that:

$$c_0 \approx_l^M s_0, c'_0 \approx_r^M s_0, c_k \approx_l^M s_k \text{ and } c'_k \approx_r^M s_k$$

It follows by Theorem 2, that $c_0 \rightarrow^k c_k$ and $c'_0 \rightarrow^k c'_k$.

From Definition 2, because $c_0 \approx_l^M s_0$ and $c'_0 \approx_r^M s_0$ we have $c_0 \simeq_L c'_0$ and because $c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$, we have $l_s = l = l'$.

Therefore the evaluation of c_k and c'_k also contains the expression $@e$, producing leakages $\texttt{bv}$ and $\texttt{bv}'$ st. $c_k\ @e \vdash \texttt{bv}$ and $c'_k\ @e \vdash \texttt{bv}'$.

From Definition 2 we have $\texttt{bv} = M(\widehat{\varphi}_{|l})$ and $\texttt{bv}' = M(\widehat{\varphi}_{|r})$, and from Eq. (4), we can deduce $\texttt{bv} \neq \texttt{bv}'$.

Therefore, we have a model M and concrete configurations $c_0 \approx_l^M s_0$, $c'_0 \approx_r^M s_0$, $c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$ such that,

$$c_0 \simeq_L c'_0 \ \wedge \ c_0 \underset{t}{\rightarrow}^k c_k \underset{\texttt{bv}}{\longrightarrow} c_{k+1} \ \wedge \ c'_0 \underset{t}{\rightarrow}^k c'_k \underset{\texttt{bv}'}{\longrightarrow} c'_{k+1} \wedge t \cdot [\texttt{bv}] \neq t' \cdot [\texttt{bv}']$$

which shows that the program is insecure.

Cases D_JUMP**,** ITE**,** STORE: The reasoning is analogous. □

C. Usability: Stubs for Input Specification

We enable the specification of high and low variables in C source code using dummy functions that are stubbed in the symbolic execution (cf. Example 3). Note that this is at the cost of portability (it can only be used around C static libraries or when in possession of the C source code). If portability is required, the user can still refer to the binary-level specification method (Section V) which relies on manual reverse-engineering to find the offsets of secrets relatively to the initial `esp`.

A call to a function `high_input_`n`(`$addr$`)`, where n is a constant value, specifies that the memory must be initialized with n secret bytes, starting at address $addr$.

Example 3 (Stub for specifying high locations). A user can write a wrapper around a function `foo` to mark its arguments as low or high as shown in Listing 3. The function `foo` can be defined in an external library which must be statically linked with the wrapper program.

```
uint8_t secret[4]; uint8_t public[4];
high_input_4(secret); //4 bytes high input
low_input_4(public);  //4 bytes low input
return foo(secret, public);
```

Listing 3: Wrapper around external function `foo`

During the symbolic execution, the function `high_input_4(secret)` is encountered, it is stubbed as:

$\texttt{@[secret+0]} := \langle \beta_0 \,|\, \beta'_0 \rangle \qquad \texttt{@[secret+1]} := \langle \beta_1 \,|\, \beta'_1 \rangle$

$\texttt{@[secret+2]} := \langle \beta_2 \,|\, \beta'_2 \rangle \qquad \texttt{@[secret+3]} := \langle \beta_3 \,|\, \beta'_3 \rangle$

where β_i, β'_i are fresh 8-bit bitvector variables.

D. Zoom on the Lucky13 Attack

Lucky 13 [50] is a famous attack exploiting timing variations in TLS CBC-mode decryption to build a Vaudenay's padding oracle attack and enable plaintext recovery [7], [50]. We do not actually mount an attack but show how to find violations of constant-time that could potentially be exploited to mount such attack.

We focus on the function `tls-cbc-remove-padding` which checks and removes the padding of the decrypted record. We extract the vulnerable version from OpenSSL-1.0.1[6] and its patch from [16]. Finally, we check that no information is leaked during the padding check by specifying the record data as private.

On the *insecure version*, BINSEC/REL accurately reports 5 violations, and for each violation, returns the address of the faulty instruction, the execution trace which can be visualized with IDA, and an input triggering the violation. For instance, on the portion of code in Listing 4, three violation are reported: two conditional statement depending on the padding length at lines 3 and 4, and a memory access depending on the padding length at line 4. For the conditional at line 3, when the length `LEN` of the record data is set to 63, BINSEC/REL returns in 0.11s the counterexample "`data_l[62]=0; data_r[62]=16`", meaning that an execution with a padding length set to 0 will take a different path that an execution with a padding length set to 16.

```
1 pad_len = rec->data[LEN-1]; // Get padding length
2 [...]
3 for (i = LEN - pad_len; i < LEN; i++)
4   if (rec->data[i] != pad_len)
5     return -1; // Incorrect padding
```

Listing 4: Padding check in OpenSSL-1.0.1

On the *secure version*, when the length LEN of the record data is set to 63, BINSEC/REL explores all the paths in 400s and reports no vulnerability.