Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure

LESLEY-ANN DANIEL, Université Paris-Saclay, CEA, List, France and imec-DistriNet, KU Leuven, Belgium
SÉBASTIEN BARDIN, Université Paris-Saclay, CEA, List, France
TAMARA REZK, Inria, France

This paper tackles the problem of designing efficient binary-level verification for a subset of information flow properties encompassing constant-time and secret-erasure. These properties are crucial for cryptographic implementations, but are generally not preserved by compilers. Our proposal 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, for bug-finding and bounded-verification of constant-time and secret-erasure, and perform extensive experiments on a set of 338 cryptographic implementations, demonstrating the benefits of our approach. Using Binsec/Rel, we also automate two prior manual studies on preservation of constant-time and secret-erasure by compilers for a total of 4148 and 1156 binaries respectively. Interestingly, our analysis highlights incorrect usages of volatile data pointer for secret erasure and shows that scrubbing mechanisms based on volatile function pointers can introduce additional register spilling which might break secret-erasure. We also discovered that gcc -O0 and backend passes of clang introduce violations of constant-time in implementations that were previously deemed secure by a state-of-the-art constant-time verification tool operating at LLVM level, showing the importance of reasoning at binary-level.

CCS Concepts:
• Security and privacy → Logic and verification; Information flow control; Cryptanalysis and other attacks.

Additional Key Words and Phrases: constant-time, secret-erasure, information-flow analysis, binary analysis, symbolic execution.

ACM Reference Format:

1 INTRODUCTION

Safety properties [1], such as buffer overflows, have been extensively studied and numerous efficient tools have been developed for their verification [2, 3, 4, 5, 6, 7, 8]. However, safety properties are properties of individual execution traces, whereas many important security properties are expressed as sets of traces—i.e., are hyperproperties [9]. In particular, information flow properties, which regulate the leakage of information from the secret inputs of a program to public outputs, relate two execution traces—i.e., are 2-hypersafety properties.

Constant-time and secret-erasure are two examples of 2-hypersafety properties that are crucial in cryptographic implementations. The constant-time programming discipline (CT) is a software-based countermeasure to timing and
microarchitectural attacks which requires the control flow and the memory accesses of the program to be independent from the secret input. Constant-time has been proven to protect against cache-based timing attacks and is widely used to secure cryptographic implementations (e.g. BearSSL, NaCL, HACL, etc). Secret-erasure (a.k.a. data scrubbing or safe erasure) requires to clear secret data (e.g. secret keys) from the memory after the execution of a critical function, for instance by zeroing the corresponding memory. It ensures that secret data do not persist in memory longer than necessary, protecting them against subsequent memory disclosure vulnerabilities.

**Problem.** These properties are generally not preserved by compilers. For example, reasoning about constant-time requires to know whether the code \( c = (x < y) - 1 \) will be compiled to branchless code or not, but this depends on the compiler version and optimization. Similarly, scrubbing operations used for secret-erasure have no effect on the result of the program and can therefore be optimized away by the dead-store-elimination pass of the compiler, as detailed in CWE-14. Moreover, these scrubbing operations do not erase secrets that have been copied on the stack by compilers.

Several CT-analysis tools have been proposed to analyze source code, or LLVM code, but leave the gap opened for violations introduced in the executable code either by the compiler or by closed-source libraries. Binary-level tools for constant-time using dynamic approaches can find bugs, but otherwise miss vulnerabilities in unexplored portions of the code, making them incomplete. Conversely, static approaches cannot report precise counterexamples—making them of minor interest when the implementation cannot be proven secure. For secret-erasure there is currently no sound automatic analyzer. Existing approaches rely on dynamic tainting or manual binary-code analysis. While there has been some work on security preserving compilers, they are not always applicable and are ineffective for detecting errors in existing binaries.

**Challenges.** Two main challenges arise in the verification of these properties:

- First, common verification methods do not directly apply because information flow properties like constant-time and secret-erasure are not regular safety properties but 2-hypersafety properties (i.e., relating two execution traces), and their standard reduction to safety on a transformed program, self-composition, is inefficient;
- Second, it is notoriously difficult to adapt formal methods to binary-level because of the lack of structure information (data and control) and the need to explicitly reason about the memory.

A technique that scales well on binary code and that naturally comes into play for bug-finding and bounded-verification is symbolic execution (SE). While it has proven very successful for standard safety properties, its direct adaptation to 2-hypersafety properties through (variants of) self-composition suffers from a scalability issue. Some recent approaches scale better, but at the cost of sacrificing either bounded-verification (by doing under-approximations) or bug-finding (by doing over-approximations).

The idea of analyzing pairs of executions for the verification of 2-hypersafety is not new (e.g. relational Hoare logic, self-composition, product programs, multiple facets). In the context of symbolic execution, it has first been coined as ShadowSE for back-to-back testing, and later as relational symbolic execution (RelSE). However, because of the necessity to model the memory, RelSE cannot be trivially adapted to binary-level analysis. In particular, representing the memory as a large array of bytes prevents sharing between executions and precise information-flow tracking, which generates a high number of queries for the constraint solver. Hence, a direct application of RelSE does not scale.

---

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

Manuscript submitted to ACM
Proposal. We restrict to a subset of information flow properties relating traces following the same path—which includes interesting security policies such as constant-time and secret-erasure. We tackle the problem of designing an efficient symbolic verification tool for constant-time and secret-erasure 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 and secret-erasure 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 [47, 48]: 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 6.3) that RelSE alone does not scale at binary-level to analyze constant-time on real cryptographic implementations. Our key technical insights are (1) to complement RelSE with dedicated optimizations offering a fine-grained information flow tracking in the memory, improving sharing at binary-level (2) to use this sharing to track secret-dependencies and reduce the number of queries sent to the solver.

Binsec/Rel can analyze about 23 million instructions in 98 min (3860 instructions per second), outperforming similar state of the art binary-level symbolic analyzers [42, 27] (cf. Table 12, page 33), while being still correct and complete.

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 5.2.1). Second, we design new simplifications for information flow analysis: untainting (Section 5.2.2) and fault-packing (Section 5.2.3). Moreover, we formally prove that our analysis is correct for bug-finding and bounded-verification of constant-time (Section 5.3) and discuss the adaptation of the guarantees to other information-flow properties in the accompanying tech report [49];
- We propose a tool named Binsec/Rel for constant-time and secret-erasure analysis. Extensive experimental evaluation (338 samples) against standard approaches (Section 6.3) shows that it can find bugs in real-world cryptographic implementations much faster than these techniques ($\times$1000 speedup) and can achieve bounded-verification when they time out, with a performance close to standard SE ($\times$2 overhead);
- In order to prove the effectiveness of Binsec/Rel, we perform an extensive analysis of constant-time at binary-level. In particular, we analyze 296 cryptographic binaries previously verified at a higher-level (incl. codes from HACL* [13], BearSSL [11], NaCL [12]), we replay known bugs in 42 samples (incl. Lucky13 [50]) and automatically generate counterexamples (Section 6.1);
- Simon et al. [15] have demonstrated that clang’s optimizations break constant-timeness of code. We extend this work in five directions—from 192 in [15] to 4148 configurations (Section 6.2): (1) we automatically analyze the code that was manually checked in [15], (2) we add new implementations, (3) we add the gcc compiler and a more recent version of clang, (4) we add 1686 and ARM, (5) we investigate the impact of individual optimizations—i.e., the -x86-cmov-converter of clang and the if-conversion passes of gcc. Interestingly, we discovered that gcc -O0 and backend passes of clang introduce violations of constant-time that cannot be detected by LLVM verification tools like ct-verif [22] even when the -x86-cmov-converter is disabled. On a positive note, we also show that, contrary to clang, gcc optimizations tend to help preserve constant-time. This study is open-source and can be easily extended with new compilers and programs;
- Finally, we build the first framework to automatically check the preservation of secret-erasure by compilers. We use it to analyze 17 scrubbing functions—including countermeasures manually analyzed in a prior study [18].
Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk

compiled with 10 compilers with different optimization levels, for a total of 1156 binaries (Section 6.4). Our analysis: (1) confirms that the main optimization affecting the preservation of secret-erasure is the dead store elimination pass (−dse), but also shows that disabling it is not always sufficient to preserve secret-erasure, (2) shows that, while some versions of scrubbing functions based on volatile data pointer are secure, it is easy to implement this mechanism incorrectly, in particular by using a volatile pointer to non-volatile data, or passing a pointer to volatile in a function call, (3) interestingly it also shows that scrubbing mechanisms based on volatile function pointers can introduce additional register spilling that might break secret-erasure with gcc −O2 and gcc −O3, (4) finally, secret-erasure mechanisms based on dedicated secure functions (i.e., explicit_bzero, memset_s), memory barriers, and weak symbols, are preserved in all tested setups. This framework is open-source and can be easily extended with new compilers and new scrubbing functions;

Discussion. 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 information-flow properties on cryptographic implementations, while our experiments demonstrate the importance of developing verification tools reasoning at binary-level. Besides constant-time and secret-erasure, the tool can be readily adapted to other 2-hypersafety properties of interest in security (e.g., cache-side channels, or variants of constant-time taking operand size into account)—as long as they restrict to pairs of traces following the same path.

Availability. We made Binsec/Rel open-source at https://github.com/binsec/rel, our experiments are available at https://github.com/binsec/rel_bench, and in particular, our studies on the preservation of constant-time and secret-erasure by compilers are available at https://github.com/binsec/rel_bench/tree/main/properties_vs_compilers.

Extension of article [51]. This paper is an extension of the article Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level [51], with the following additional contributions:

- The leakage model considered in [51] restricts to constant-time while this work encompasses a more general subset of information flow properties. In particular, we define a new leakage model and property to capture the notion of secret-erasure (cf. Sections 4.2 and 4.3);
- We extend the Binsec/Rel tool to verify the secret-erasure property;
- We perform an experimental evaluation on the preservation of secret-erasure by compilers (cf. Section 6.4). This evaluation highlights incorrect usages of volatile data pointers for secret erasure, and shows that scrubbing mechanisms based on volatile function pointers can introduce additional violations from register spilling;
- Using Binsec/Rel, we also investigate the role of individual compiler optimizations in the preservation of secret-erasure and constant-time. For constant-time, we show that the if-conversion passes of gcc may help enforce constant-time in ARM binaries. We also show that disabling the cmov-converter is not always sufficient to preserve constant-time in the backend-passes of clang. For secret-erasure, we confirm the key role of the dead store elimination pass (−dse), but also show that disabling it does not always preserve secret-erasure.

In addition, we provide a technical report [49] which contains full proofs of relative completeness and correctness of the analysis, contains an evaluation of the scalability of Binsec/Rel according to the size of the input, and details the vulnerabilities introduced by clang with examples.
2 BACKGROUND

In this section, we present the basics of information flow properties and symbolic execution. Small examples of constant-time and standard adaptations of symbolic execution are presented in Section 3, while formal definitions of information flow policies (including constant-time and secret-erasure) are given in Section 4.

Information flow properties. Information flow policies regulate the transfer of information between public and secret domains. To reason about information flow, the program input is partitioned 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 [52]). Constant-time and secret-erasure can be expressed as information flow properties. Constant-time requires both the program control flow and the memory accesses to be independent from high input. It protects against timing and microarchitectural attacks (exploiting cache, port contention, branch predictors, etc.). Secret-erasure requires specific memory locations (typically the call stack) to be independent from high input when returning from a critical function. It ensures that secret data do not persist in memory longer than necessary [53], protecting these secret data against subsequent memory exposure, e.g. memory disclosure vulnerabilities, access to persistent storage (swap memory).

Contrary to standard safety properties which state that nothing bad can happen along one execution trace, information flow properties relate two execution traces—they are 2-hypersafety properties [9]. Unfortunately, the vast majority of symbolic execution tools [3, 2, 8, 54, 55, 56] 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].

Symbolic execution. Symbolic Execution (SE) [57, 37, 3] consists in executing a program on symbolic inputs instead of concrete inputs. Variables and expressions of the program are represented as terms over these 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 [58], to generate a concrete input exercising the path. Combining these two steps, SE can explore 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 progress in program analysis and constraint solving over the last two decades have made SE a tool of choice for intensive testing [38, 37], vulnerability analysis [8, 59, 60] and other security-related analysis [61, 62].

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.

Binary-level code analysis is notoriously more challenging than source code analysis [36, 35]. First, evaluation and assignments of source code variables become memory load and store operations, requiring to reason explicitly 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 indirect 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 [3, 8, 54, 55, 56, 62, 63]. In this paper, we build on top of the binary-analysis platform Binsec [64] and in particular its symbolic execution engine Binsec/SE [56].

One of the key components of binary-level symbolic execution is the representation of the memory. A first solution, adopted in Binsec/SE [56] and Bap [65], is to use a fully symbolic memory model in which the memory is represented as a symbolic array of bytes. Other solutions consist in concretizing (parts of) the memory. For instance, angr [55] uses a partially symbolic memory model [8] in which write addresses are concretized and symbolic loads are encoded as symbolic if-the-else expressions. Fully symbolic memory models incur a performance overhead compared to partially symbolic (or concrete) memory models. However, they can model all possible values that load/write addresses can take—instead of considering only a subset of the possible addresses. Hence, they offer better soundness guarantees and are better suited for bounded-verification.

**Logical notations.** Binsec/SE relies on the theory of bitvectors and arrays, QF_ABV [66]. Values (e.g. registers, memory addresses, memory content) are modeled with fixed-size bitvectors [67]. We use the type $B_{\nu m}$, where $m$ is a constant number, to represent symbolic bitvector expressions of size $m$. The memory is modeled with a logical array of type $(\text{Array } B_{\nu 32} B_{\nu 8})$ (assuming a 32 bit architecture). A logical array is a function $(\text{Array } IV)$ that maps each index $i \in I$ to a value $v \in V$. Operations over arrays are:

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

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

- $\text{select } (\text{store } a i v) i = v$: a store of a value $v$ at index $i$ followed by a select at the same index returns the value $v$;
- $i \neq j \implies \text{select } (\text{store } a i v) j = \text{select } a j$: a store at index $i$ does not affect values stored at other indexes $j$.

3 **MOTIVATING EXAMPLE: CONSTANT-TIME ANALYSIS**

Consider the constant-time policy applied to the toy program in Listing 1. The outcome of the conditional instruction 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. Constant-time holds for a program if there is no insecure leak.

```
1 x = secret_input();
2 y = public_input();
3 if (y != 0) return 0; // leak y = 0
4 return tab[z]; // leak z
```

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

```
1 store ebp-8 $\langle\beta | \beta'\rangle$ // store high input
2 store ebp-4 $y$ // store low input
3 eax := load ebp-4 // assign $\lambda$ to eax
4 ite eax ? 1 : 2 // leak $\lambda = 0$
5 [...]
```

Listing (2) Compiled version of Listing 1, where $\langle\beta | \beta'\rangle$ (resp. $\langle\lambda\rangle$) denotes a high (resp. low) input.

**Example.** Consider 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.

Manuscript submitted to ACM
because $y = y'$; hence this leak does not depend on the secret input and is secure. On the contrary, the leakages $x$ and $x'$ at line 4 can differ in both executions (e.g. with $x = 0$ 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.

### 3.1 Symbolic Execution and Self-Composition (SC)

Symbolic execution can be adapted to the case of constant-time, 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 [69]. 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. Additionally, insecurity queries specific to constant-time are sent before each control-flow instruction and memory access to determine whether they depend on the secret—if an insecurity query is satisfiable then a constant-time 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 jump (line 3), resulting in the formula: $\lambda = \beta \land y = \lambda \land c = (\lambda \neq 0)$. Second, self-composition is applied on the formula with precondition $\lambda = \lambda'$ to constrain the low inputs to be equal in both executions. Finally, a postcondition $c \neq c'$ asks whether the value of the condition can differ, resulting in the following insecurity query:

$$
\lambda = \lambda' \land (x = \beta \land y = \lambda \land c = (\lambda \neq 0) \land x' = \beta' \land y' = \lambda' \land c' = (\lambda' \neq 0)) \land c \neq c'$$

This formula is sent to an SMT-solver. If the solver returns unsat, meaning that the query is not satisfiable, then the condition does not differ in both executions and thus is secure. Otherwise, it means that the outcome of the condition depends on the secret and the solver returns a counterexample satisfying the insecurity query. Here, the SMT-solver Z3 [70] 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.

**Limits.** Basic self-composition suffers from two weaknesses:

- It generates insecurity queries at each control-flow instruction and memory access. Yet, as seen in the previous example, insecurity queries could be spared when expressions do not depend on secrets.
- 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 which only depend on public input are equal in both executions, the self-composed formula contains redundancies that are not exploited.

### 3.2 Relational Symbolic Execution (ReSE)

ReSE improves over self-composition by maximizing sharing between the pairs of executions [47, 48]. ReSE models two executions of a program $P$ in the same symbolic execution instance, let us call them $p$ and $p'$. During ReSE, variables of $P$ are mapped to relational expressions which are either pairs of expressions or simple expressions. Variables that must be equal in $p$ and $p'$ (i.e., the low inputs) are represented as simple expressions whereas those that may be different
way) through symbolic execution using these relational expressions: if the evaluation of an expression only involves pairs (i.e., the secret input) are represented as of queries compared to SC Curve25519-donna [71]. Both SC and RelSE fail to prove the program secure in less than 1h. RelSE does reduce the number of queries compared to SC, but it is not sufficient.

### 3.3 Challenge of binary-level analysis

Recall that, Binsec/SE represents the memory as a special variable of type (Array $\mathbb{B}_{32} \times \mathbb{B}_{8}$). Consequently, it is not possible to directly store relational expressions in it. 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. 2. 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.

**Practical impact.** Table 1 reports the performance of constant-time analysis on an implementation of elliptic curve Curve25519-donna [71]. Both SC and RelSE fail to prove the program secure in less than 1h. RelSE does reduce the number of queries compared to SC, but it is not sufficient.
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 5.2). In the particular example of Table 1, our prototype Binsec/Rel proves 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 explored per second.

4 CONCRETE SEMANTICS AND LEAKAGE MODEL
We present the leakage models in an intermediate language called Dynamic Bitvectors Automatas (DBA) [72].

4.1 Dynamic Bitvectors Automatas

DBA [72], shown in Fig. 3, is the representation used in Binsec [56] to model programs and perform its analysis.

![Fig. 3. The syntax of DBA programs, where $l, l_1$ and $l_2$ are program locations, $v$ is a variable and $bv$ is a value.](image)

Let $Inst$ denote the set of instructions and $Loc$ the set of program locations. A program $P : Loc \rightarrow Inst$ 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:

- $l \in Loc$ is the current location, and $P[l]$ returns the current instruction,
- $r : Var \rightarrow BV_n$ is a register map that maps variables to their bitvector value,
- $m : BV_{32} \rightarrow BV_8$ is the memory, mapping 32-bit addresses to bytes and accessed by operators $load$ and $store$.

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. Let $Loc_\perp \subseteq Loc$ the set of halting program locations such that $l \in Loc_\perp \iff P[l] = \text{halt}$. For the evaluation of indirect jumps, we define a partial one-to-one correspondence from bitvectors to program locations, $to_{loc} : BV_{12} \rightarrow Loc$. If a bitvector corresponds to an illegal location (e.g. non-executable address), $to_{loc}$ is undefined.
4.2 Leakage Model

The behavior of programs is modeled with an instrumented operational semantics in which each transition is labeled with an explicit notion of leakage. Building on Barthe, Grégoire, and Laporte’s framework [73], the semantics is parameterized with leakage functions, which permits to consider several leakage models.

The set of program leakages, denoted $\mathcal{L}$, is defined according to the leakage model. A transition from a configuration $c$ to a configuration $c'$ produces a leakage $t \in \mathcal{L}$, denoted $c \xrightarrow{t} c'$. Analogously, the evaluation of an expression $e$ in a configuration $(l, r, m)$, produces a leakage $t \in \mathcal{L}$, denoted $(l, r, m) \xrightarrow{t} bv$. The leakage of a multistep execution is the concatenation of leakages, denoted $\cdot$, produced by individual steps. We use $\xrightarrow{t}^k$ with $k$ a natural number to denote $k$ steps in the concrete semantics.

<table>
<thead>
<tr>
<th>Expr</th>
<th>Instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>CST $(l, r, m) \xrightarrow{v_e} bv$</td>
<td>HALT $P[l] = \text{halt}$ $(l, r, m)$ $\xrightarrow{\lambda_e(l)} (l, r, m)$</td>
</tr>
<tr>
<td>VAR $(l, r, m) \xrightarrow{v_e} r \xrightarrow{v}$</td>
<td>S_JUMP $P[l] = \text{goto } l'$ $(l, r, m) \xrightarrow{\lambda_{pc}(l')} (l', r, m)$</td>
</tr>
<tr>
<td>BINOP $(l, r, m) e_1 \xrightarrow{t_1} bv_1 \xrightarrow{t_2} bv_2$</td>
<td>D_JUMP $P[l] = \text{goto } e$ $(l, r, m) e \xrightarrow{t} bv$ $l' \xrightarrow{=} \text{to} _ \text{loc}(bv)$ $(l, r, m) \xrightarrow{t \cdot \lambda_{pc}(l')} (l', r, m)$</td>
</tr>
<tr>
<td>LOAD $(l, r, m) e \xrightarrow{t} bv$</td>
<td>ITE-TRUE $P[l] = \text{ite } e \xrightarrow{?} l_1 : l_2$ $(l, r, m) e \xrightarrow{t} bv$ $bv \neq 0$ $(l, r, m) \xrightarrow{t \cdot \lambda_{pc}(l_1)} (l_1, r, m)$</td>
</tr>
<tr>
<td>UNOP $(l, r, m) e \xrightarrow{t} bv$</td>
<td>ITE-FALSE $P[l] = \text{ite } e \xrightarrow{?} l_1 : l_2$ $(l, r, m) e \xrightarrow{t} bv$ $bv = 0$ $(l, r, m) \xrightarrow{t \cdot \lambda_{pc}(l_2)} (l_2, r, m)$</td>
</tr>
<tr>
<td>$(l, r, m) \xrightarrow{t \cdot \lambda_{pc}(l_2)} (l_2, r, m)$</td>
<td>ASSIGN $P[l] = v := e$ $(l, r, m) e \xrightarrow{t} bv$ $(l, r, m) \xrightarrow{t} (l + 1, r [v \mapsto bv], m)$</td>
</tr>
<tr>
<td>$(l, r, m) e \xrightarrow{t} bv$</td>
<td>STORE $P[l] = \text{store } e e'$ $(l, r, m) e \xrightarrow{t} bv$ $(l, r, m) e' \xrightarrow{t'} bv'$ $(l, r, m) \xrightarrow{t' \cdot t \cdot \lambda_{pc}(bv) \cdot \lambda_{pc}(bv')} (l + 1, r, m [bv \mapsto bv'])$</td>
</tr>
</tbody>
</table>

Fig. 4. Concrete evaluation of DBA instructions and expressions.
The concrete semantics is given in Fig. 4 and is parameterized with leakage functions \( \lambda_4 : BV \rightarrow \mathcal{L} \), \( \lambda : BV \times BV \rightarrow \mathcal{L} \), \( \lambda_3 : BV_{32} \rightarrow \mathcal{L} \), \( \lambda_{pc} : Loc \rightarrow \mathcal{L} \), \( \lambda_\perp : Loc \rightarrow \mathcal{L} \), \( \lambda_\mu : BV_{32} \times BV_8 \rightarrow \mathcal{L} \). A leakage model is an instantiation of the leakage functions. We consider the program counter, memory obliviousness, size noninterference and constant-time, leakage models defined in [73]. In addition, we define the operand noninterference and secret-erasure leakage models.

**Program counter** [73]. The programs counter leakage model leaks the control flow of the program. The leakage of a program is a list of program location: \( \mathcal{L} \triangleq \text{List} (\text{Loc}) \). The outcome of conditional jumps and the address of indirect jumps is leaked: \( \lambda_{pc}(l) = [l] \). Other instructions produce an empty leakage.

**Memory obliviousness** [73]. The memory obliviousness leakage model leaks the sequence of memory addresses accessed along the execution. The leakage of a program is a list of 32-bit bitvectors representing addresses of memory accesses: \( \mathcal{L} \triangleq \text{List}(BV_{32}) \). The addresses of memory load and stores are leaked: \( \lambda_\perp(e) = [e] \). Other instructions produce an empty leakage.

**Operand noninterference.** The operand noninterference leakage model leaks the value of operands (or part of it) for specific operators that execute in non constant-time. The leakage of a program is a list of bitvector values: \( \mathcal{L} \triangleq \text{List}(BV) \). Functions \( \lambda_4 \) and \( \lambda_\mu \) are defined according to architecture specifics. For instance, in some architectures, the execution time of shift or rotation instructions depends on the shift or rotation count\(^3\). In this case, we can define \( \lambda_{<<}(bv_1, bv_2) = [bv_2] \). Other instructions produce an empty leakage.

**Size noninterference** [73]. The size noninterference leakage model is a special case of operand noninterference where the size of the operand is leaked. For instance, knowing that the execution time of the division depends on the size of its operands, we can define \( \lambda_\mu(bv, bv') = [size(bv), size(bv')] \).

**Constant-time** [73]. The constant-time leakage model combines the program counter and the memory obliviousness security policies. The set of leakage is defined as \( \mathcal{L} \triangleq \text{List} (\text{Loc} \cup BV_{32}) \). The control flow is leaked \( \lambda_{pc}(l) = [l] \), as well as the memory accesses \( \lambda_\perp(e) = [e] \). Other instructions produce an empty leakage. Note that some definitions of constant-time also include size noninterference [73] or operand noninterference [11].

**Secret-erasure.** The secret-erasure leakage model leaks the index and value of every store operation—values that are overwritten are filtered-out from the leakage trace (as we formalize later in Definition 3). With regard to secret dependent control-flow, we define a conservative notion of secret-erasure forbidding to branch on secrets—thus including the program counter policy. The leakage of a program is a list of locations and pairs of bitvector values: \( \mathcal{L} \triangleq \text{List}(\text{Loc} \cup (BV_{32} \times BV_8)) \). The control flow is leaked \( \lambda_{pc}(l) = [l] \), as well as the end of the program \( \lambda_\perp(l) = [l] \), and the list of store operations \( \lambda_\mu((bv, bv')) = [(bv, bv')] \). Other instructions produce an empty leakage.

4.3 Secure program

Let \( H_h \subseteq Var \) be the set of high (secret) variables and \( L_p = Var \setminus H_h \) be the set of low (public) variables. Analogously, we define \( H_\perp \subseteq BV_{32} \) (resp. \( L_\perp = BV_{32} \setminus H_\perp \)) as the addresses containing high (resp. low) input in the initial memory. The low-equivalence relation over concrete configurations \( c \) and \( c' \), denoted \( c \equiv_L c' \), is defined as the equality of low variables and low parts of the memory. Formally, two configurations \( c \equiv (l, r, m) \) and \( c' \equiv (l', r', m') \) are low-equivalent if and only if for all variable \( v \in L_p, r v = r' v \) and for all address \( a \in L_\perp, m a = m' a \).

\(^3\)See https://bearsll.org/constanttime.html
Security is expressed as a form of observational noninterference that is parameterized by the leakage model. Intuitively it guarantees that low-equivalent configurations produce the same observations, according to the leakage model:

**Definition 1 (Observational noninterference (ONI)).** A program is observationally noninterferent if and only if for all low-equivalent initial configurations $c_0$ and $c_0'$, and for all $k \in \mathbb{N}$,

$$c_0 \equiv_L c_0' \land c_0 \xrightarrow{t}^k c_k \land c_0' \xrightarrow{t'}^k c_k' \implies \text{filter}(t) = \text{filter}(t')$$

The property is parameterized by a function, $\text{filter} : L \to L$, that further restricts the leakage.

**Definition 2 (Constant-time).** A program is constant-time (CT) if it is ONI in the constant-time leakage model with filter set to the identity function.

**Definition 3 (Secret-erasure).** A program enforces secret-erasure if it is ONI in the secret-erasure leakage model with filter set to the identity function for control-flow leakages and only leaking store values at the end of the program ($l \in \text{Loc}_L$), restricting to values that have not been overwritten by a more recent store. Formally, $\text{filter}(t) = \text{filter}'(t, m_e)$ where $m_e$ is the empty partial function from $\text{BV}_{128}$ to $\text{BV}_8$ and $\text{filter}'(t, m_{acc})$ is defined as:

- **FILTER-EMPTY** $\text{filter}'(\epsilon, m_{acc}) = \epsilon$
- **FILTER-STORE** $\text{filter}'((a, \nu) \cdot t, m_{acc}) = \text{filter}'(t, m_{acc}[a \mapsto \nu])$
- **FILTER-CF** $l \notin \text{Loc}_L \quad \text{filter}'(l \cdot t, m_{acc}) = l \cdot \text{filter}'(t, m_{acc})$
- **FILTER-HALT** $a_k \in \text{dom}(m_{acc}) \quad l \in \text{Loc}_L \quad \text{filter}'(l \cdot t, m_{acc}) = m_{acc}(a_0) \cdot \ldots \cdot m_{acc}(a_n)$

Intuitively, $m_{acc}$ is a function used to accumulate values written to the memory and leak them at the end of a program. The **FILTER-STORE** rule accumulates a store operation $(a, c)$ from the leakage trace into the function $m_{acc}$. Notice that because $m_{acc}$ is a function, if $m_{acc}(a)$ is already defined, its value will be replaced by $\nu$ after $m_{acc}[a \mapsto \nu]$. The **FILTER-CF** rule adds control-flow label to the final leakage trace. Finally, the **FILTER-HALT** rule is evaluated when a final location is reached and leaks all the store values accumulated in $m_{acc}$. For example, $\text{filter}((a, x) \cdot (b, y) \cdot (a, z) \cdot l_\bot)$ where $l_\bot \in \text{Loc}_L$ will return the leakage $y \cdot z$.

### 5 Binary-Level Relational Symbolic Execution

Binary-level symbolic execution relies on the quantifier-free theory of fixed-size bitvectors and arrays (QF_ABV [66]). We let $\beta, \beta', \lambda, \varphi$, range over the set of formulas $\Phi$ in the QF_ABV logic. A **relational** formula $\varphi$ is either a QF_ABV formula ($\varphi$) or a pair $(\varphi_l | \varphi_r)$ of two QF_ABV formulas. We denote $\varphi_l$ (resp. $\varphi_r$), the projection on the left (resp. right) value of $\varphi$. If $\varphi = (\cdot)$, then $\varphi_l$ and $\varphi_r$ are both defined as $\varphi$. Let $\Phi$ be the set of relational formulas and $\text{BV}_n$ be the set of relational symbolic bitvectors of size $n$.

**Symbolic configuration.** Our symbolic evaluation restricts to pairs of traces following the same path—which is sufficient for constant-time and our definition of secret-erasure. Therefore, a symbolic configuration only needs to consider a single program location $l \in \text{Loc}$ at any point of the execution. A **symbolic configuration** is of the form $(l, \rho, \mu, \pi)$ where:

- $l \in \text{Loc}$ is the current program point,
- $\rho : \text{Var} \to \Phi$ is a symbolic register map, mapping variables from a set $\text{Var}$ to their symbolic representation as a relational formula in $\Phi$. 

Manuscript submitted to ACM
• $\hat{\mu} : (\text{Array } B_{32} B_{64}) \times (\text{Array } B_{32} B_{64})$ is the symbolic memory—a pair of arrays of values in $B_{64}$ indexed by addresses in $B_{32}$,
• $\pi \in \Phi$ is the path predicate—a conjunction of conditional statements and assignments encountered along a path.

Symbolic evaluation of instructions, denoted $s \xrightarrow{} s'$ where $s$ and $s'$ are symbolic configurations, is given in Figure 5. The evaluation of an expression $expr$ to a relational formula $\phi$, is denoted $(\rho, \mu, \pi) \ xpr \mapsto \phi$. A model $M$ assigns concrete values to symbolic variables. The satisfiability of a formula $\pi$ with a model $M$ is denoted $M \models \pi$. In the implementation, an SMT-solver is used to determine satisfiability of a formula and obtain satisfying model, denoted $M \models_{\text{smt}} \pi$. Whenever the model is not needed for our purposes, we leave it implicit and simply write $\models \pi$ or $\models_{\text{smt}} \pi$ for satisfiability.

The symbolic evaluation is parameterized by symbolic leakage predicates $\tilde{\lambda}, \tilde{\lambda}_0, \tilde{\lambda}_d, \tilde{\lambda}_d^j, \tilde{\lambda}_{\text{ite}}$ and $\tilde{\lambda}_\perp$ which are instantiated according to the leakage model (details on the instantiation will be given in Section 5.1.2). Symbolic leakage predicates take as input a path predicate and expressions that can be leaked, and return $\text{true}$ if and only if no secret data can leak. The rules of the symbolic evaluation are guarded by these symbolic leakage predicates: a rule can only be evaluated if the associated leakage predicate evaluates to $\text{true}$, meaning that no secret can leak. If a symbolic leakage predicate evaluates to $\text{false}$ then a secret leak is detected and the analysis is stuck. Detailed explanations of (some of) the symbolic evaluation rules follow:

- **cst** is the evaluation of a constant $bv$ and returns the corresponding symbolic bitvector as a simple expression $\langle bv \rangle$.
- **load** is the evaluation of a load expression. It returns a pair of logical $\text{select}$ formulas from the pair of symbolic memories $\hat{\mu}$ (the box in the hypotheses should be ignored for now, it will be explained in Section 5.2). Note that the returned expression is always duplicated as the $\text{select}$ must be performed in the left and right memories independently.
- **d_jump** is the evaluation of an indirect jump. It finds a concrete value $l'$ for the jump target, and updates the path predicate and the next location. Note that this rule is nondeterministic as $l'$ can be any concrete value satisfying the path constraint.
- **ite-true** is the evaluation of a conditional jump when the expression evaluates to $\text{true}$ (the $\text{false}$ case is analogous). If the condition guarding the $\text{true}$-branch is satisfiable, the rule updates the path predicate and the next location to explore it.
- **assign** is the evaluation of an assignment. It allocates a fresh symbolic variable to avoid term-size explosion, and updates the register map and the path predicate. The content of the box in the hypothesis and the rule canonical-assign should be ignored for now and will be explained in Section 5.2.
- **store** is the evaluation of a store instruction. It evaluates the index and value of the store and updates the symbolic memories and the path predicate with a logical store operation.

5.1 Security evaluation

For the security evaluation, we start by defining a general predicate, $secLeak$, which takes as an input a path predicate and a relational expression that is leaked, and returns $\text{true}$ if and only if no secret data can leak (cf. Section 5.1.1). Then, we use this $secLeak$ predicate to instantiate symbolic leakage predicates $\tilde{\lambda}, \tilde{\lambda}_0, \tilde{\lambda}_d, \tilde{\lambda}_d^j, \tilde{\lambda}_{\text{ite}}$ and $\tilde{\lambda}_\perp$ according to the leakage model (cf. Section 5.1.2).
Fig. 5. Symbolic evaluation of DBA instructions and expressions where fresh returns a fresh symbolic variable, canonical(\(\hat{\varphi}\)) is true if \(\hat{\varphi}\) is in canonical form; and \(\epsilon\) (resp. \(\epsilon\)) is the logical counterpart of the concrete operator \(\epsilon\) (resp. \(\epsilon\)), lifted to relational expressions.
5.1.1 Predicate secLeak. We define a predicate \( \text{secLeak} : \Phi \times \Phi \rightarrow \text{Bool} \) which ensures that a relational formula does not differ in its right and left components, meaning that it can be leaked securely:

\[
\text{secLeak}(\varphi, \pi) = \begin{cases} 
  \text{true} & \text{if } \varphi \equiv \langle \varphi \rangle \\
  \text{true} & \text{if } \varphi \equiv \langle \varphi_l | \varphi_r \rangle \land \not\exists \text{smt } \pi \land \varphi_l \neq \varphi_r \\
  \text{false} & \text{otherwise}
\end{cases}
\]

By definition, 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. However, a duplicated expression \( \langle \varphi_l | \varphi_r \rangle \) may depend on secrets. Hence an insecurity query must be sent to the solver to ensure that the leak is secure.

5.1.2 Instantiation of leakage predicates. Symbolic leakage predicates are instantiated according to the concrete leakage models defined in Section 4.2. Note that the analysis can be easily be extended to other leakage models by defining symbolic leakage predicates accordingly.

Program counter. Symbolic leakage predicates ensure that the outcome of control-flow instructions and the addresses of indirect jumps are the same in both executions:

\[
\hat{\lambda}_{d\ell}(\pi, \varphi) = \text{secLeak}(\varphi, \pi) \quad \text{and} \quad \hat{\lambda}_{ite}(\pi, \varphi) = \text{secLeak}(\text{eq}_0 \varphi, \pi)
\]

where \( \text{eq}_0 : B \rightarrow B \) returns \( \text{true} \) if \( x = 0 \) and \( \text{false} \) otherwise, and \( \text{eq}_0 \) is the lifting of \( \text{eq}_0 \) to relational formulas. Other symbolic leakage predicates evaluate to true.

Memory obliviousness. Symbolic leakage predicates ensure that store and load indexes are the same in both executions:

\[
\hat{\lambda}_@ (\pi, \varphi) = \text{secLeak}(\varphi, \pi)
\]

Other symbolic leakage predicates evaluate to true.

Operand noninterference. Symbolic leakage predicates ensure that operands (or part of them) are the same in both executions for specific operators that execute in non constant-time. For instance, for architectures in which the execution time of shift depends on the shift count,

\[
\hat{\lambda}_{\ll}(\pi, \varphi, \phi) = \text{secLeak}(\varphi, \pi)
\]

Other symbolic leakage predicates evaluate to true.

Size noninterference (special case of operand noninterference). Symbolic leakage predicates ensure that the size of operands is the same in both executions for specific operators that execute in non constant-time. For instance for the division, we have

\[
\hat{\lambda}_{\div}(\pi, \varphi, \phi) = \text{secLeak}(\text{size } \varphi, \pi)
\]

where \( \text{size} : B \rightarrow B \) is a function that returns the size of a symbolic bitvector and \( \text{size} \) its lifting to relational expressions. Other symbolic leakage predicates evaluate to true.

Constant-time. This policy is a combination of the program counter and the memory obliviousness policies. Symbolic leakage predicates \( \hat{\lambda}_{d\ell} \) and \( \hat{\lambda}_{ite} \) are defined like in the program counter policy, while \( \hat{\lambda}_@ \) is defined like in the memory obliviousness policy. Other symbolic leakage predicates evaluate to true.

Secret-erasure. At the end of the program, a symbolic leakage predicate ensures that the parts of memory that have been written by the program are the same in both executions:

\[
\hat{\lambda}_\bot(\pi, \mu) = \bigwedge_{i \in \text{addr}(\mu)} \text{secLeak}(\text{select}(\mu | i), \pi)
\]

where \( \text{addr}(\mu) \) is the list of store indexes in \( \mu \).

5.1.3 Specification of high and low input. By default, the content of the memory and registers is low so the user has to specify memory addresses that initially contain secret inputs. Addresses of high variables can be specified as offsets.
from the initial stack pointer esp (which requires manual reverse engineering), or using dummy functions to annotate secret variables at source level (which is easier but only applies to libraries or requires access to source code).

5.1.4 Bug-finding. A vulnerability is found when the function $secLeak(\varphi, \pi)$ evaluates to $false$. In this case, the insecurity query is satisfiable and the solver returns a model $M$ such that $M \models_{\text{SMT}} \pi \land (\varphi_{l} \neq \varphi_{r})$. The model $M$ assigns concrete values to variables that satisfy the insecurity query. Therefore it can be returned as a concrete counterexample that triggers the vulnerability, along with the current location of the vulnerability.

5.2 Optimizations for binary-level symbolic execution

Relational symbolic execution does not scale in the context of binary-level analysis (see RelSE in Table 5). In order to achieve better scalability, we enrich our analysis with an optimization, called on-the-fly-read-over-write (FlyRow in Table 6), based on read-over-write [74]. 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 6), specifically targeting RelSE for information flow analysis.

5.2.1 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 [74]. Read-over-write (Row) [74] is a simplification for the theory of arrays that efficiently resolves $select$ operations. It is particularly efficient in the context of binary-level analysis where the memory is represented as an array and formulas contain many $store$ and $select$ operations. The standard read-over-write optimization [74] 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 [74], we show in Section 6.3.3 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 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 performance of the analysis.

Memory Lookup. The symbolic memory can be seen as the history of the successive $store$ operations beginning with the initial memory $\mu_0$. Therefore, a memory $select$ can be resolved by going back up the history and comparing the index to load, with indexes previously stored. Our FlyRow optimization consists in replacing selection in the memory (Figure 5, Load rule, boxed hypothesis) by a new function $lookup : ((\text{Array } \mathbb{B}32, \mathbb{B}8)) \times ((\text{Array } \mathbb{B}32, \mathbb{B}8)) \times \mathbb{B}32 \rightarrow \mathbb{B}8$ which takes a relational memory and a relational index, and returns the relational bitvector value stored at that index. For simplicity we define the function for simple indexes and detail the lifting to relational indexes in the companion
technical report [49]:

\[
\text{lookup}(\mu_0, i) = \langle \text{select}(\mu_0[i], t) | \text{select}(\mu_0[r], t) \rangle
\]

\[
\text{lookup}(\mu_n, i) = \begin{cases} 
\langle \varphi_i \rangle & \text{if } \text{eq}^\#(i, \kappa) \land \text{eq}^\#(\varphi_l, \varphi_r) \\
\langle \varphi_i | \varphi_r \rangle & \text{if } \text{eq}^\#(i, \kappa) \land \neg \text{eq}^\#(\varphi_l, \varphi_r) \\
\text{lookup}(\mu_{n-1}, i) & \text{if } \neg \text{eq}^\#(i, \kappa) \\
\langle \text{select}(\mu_n[i], t) | \text{select}(\mu_n[r], t) \rangle & \text{if } \text{eq}^\#(i, \kappa) = \bot 
\end{cases}
\]

where \(\mu_n \triangleq \langle \text{store}(\mu_{n-1}[i], \kappa, \varphi_l) | \text{store}(\mu_{n-1}[r], \kappa, \varphi_r) \rangle\)

where \(\text{eq}^\#(i, \kappa)\) is a comparison function relying on syntactic term equality, which returns true (resp. false) only if \(i\) and \(\kappa\) 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:

\[
\mu = \begin{array}{c}
\text{ebp - 4} \langle \lambda \rangle \\
\text{ebp - 8} \langle \beta | \beta' \rangle \\
\text{exp} \langle \text{ebp} \rangle \\
\end{array}
\]

- A call to \(\text{lookup}(\mu, \text{ebp - 4})\) returns \(\lambda\).
- A call to \(\text{lookup}(\mu, \text{ebp - 8})\) first compares the indexes \([\text{ebp - 4}]\) and \([\text{ebp - 8}]\). 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 \((\beta | \beta')\).
- A call to \(\text{lookup}(\mu, \text{esp})\) tries to compare the indexes \([\text{ebp - 4}]\) and \([\text{esp}]\). Without further information, the equality or disequality of \(\text{ebp}\) and \(\text{esp}\) cannot be determined, therefore the lookup is aborted and the \text{select} operation cannot be simplified.

Term rewriting. To improve the conclusiveness of syntactic equality checks for the read-over-write, the terms are assumed to be in normalized form \(\beta + o\) where \(\beta\) is a base (i.e., an expression on symbolic variables) and \(o\) is a constant offset. The comparison of two terms \(\beta + o\) and \(\beta' + o'\) in normalized form can be efficiently computed as follows: if the bases \(\beta\) and \(\beta'\) are syntactically equal, then return \(o = o'\), otherwise the terms are not comparable. In order to apply FlyRow, we normalize all the formulas created during the symbolic execution using rewriting rules similar as those defined in [74]. An excerpt of these rules is given in Fig. 6. Intuitively, these rewriting rules put symbolic variables at the beginning of the term and the constants at the end (see Example 2).

\[
\text{normalize } (c + t) = t + c \quad \text{normalize } (c + c') = (c + c') \\
\text{normalize } ((t + c) + c') = t + (c + c') \quad \text{normalize } ((t + c) + t') = (t + t') + c \\
\text{normalize } ((t + c) - (t' + c')) = (t + t') + (c - c') \quad \text{normalize } ((t + c) - (t' + c')) = (t - t') + (c - c')
\]

Fig. 6. Rewriting rules for normalization (non-exhaustive). All expressions belong to the set \(\mathbb{B}o\) where \(c, c'\) are bitvector constants and \(t, t'\) are arbitrary bitvector terms. Note that \((c + c')\) is a constant value, not a term.

Example 2 (Normalized formula). \(\text{normalize } ((eax + 4) + (ebx + 4)) = (eax + ebx) + 8\)

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 \(x + o\) where...
x is a bitvector variable, and o is a constant bitvector offset, and we only inline formulas that are in canonical form (see rule _canonical-assign_ in Fig. 5). It enables rewriting of most of the memory accesses on the stack which, are of the form ebp + bv, while avoiding term-size explosion.

5.2.2 Untainting. After the evaluation of a rule with the predicate secLeak for a duplicated expression ⟨𝜙₁ | 𝜙ᵣ⟩, we know that the equality 𝜙₁ = 𝜙ᵣ 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 ⟨𝑥ᵢ + 1 | 𝑥ᵣ + 1⟩, where 𝑥ᵢ and 𝑥ᵣ are symbolic variables. If the leak is secure, we can deduce that 𝑥ᵢ = 𝑥ᵣ and replace all occurrences of 𝑥ᵣ by 𝑥ᵢ in the rest of the symbolic execution.

We define in Fig. 7 a function _untaint_ which takes a register map 𝜌, a memory 𝜇, and a duplicated expression 𝜙. It deduces variable equalities from 𝜙, propagate them in 𝜌 and 𝜇, and returns a pair of updated register map and memory (ρ’, 𝜇’). Intuitively, if the equality of variables 𝑥ᵢ and 𝑥ᵣ can be deduced from secLeak(𝜙, 𝜋), the _untaint_ function replaces occurrences of 𝑥ᵣ by 𝑥ᵢ in the memory and the register map. As a result, a duplicated expression ⟨𝑥ᵢ | 𝑥ᵣ⟩ would be replaced by the simple expression ⟨𝑥ᵢ⟩ in the rest of the execution⁴.

\[
\begin{align*}
\text{untaint}(\rho, \hat{\mu}, x) &= (\rho[x_i, x], \hat{\mu}[x_i]) & \text{untaint}(\rho, \hat{\mu}, t) &= (\rho[t_l | t_r]) \\
\text{untaint}(\rho, \hat{\mu}, t) &= \text{untaint}(\rho, \hat{\mu}, t_l) & \text{untaint}(\rho, \hat{\mu}, t) &= \text{untaint}(\rho, \hat{\mu}, t_r)
\end{align*}
\]

Fig. 7. Untainting rules where 𝑥ᵢ, 𝑥ᵣ are bitvector variables of the same size, 𝑡ᵢ, 𝑡ᵣ, 𝑘, 𝑙 are bitvector terms such that 𝑡ᵢ, 𝑡ᵣ, 𝑘 have the same size, :: indicates the concatenation of bitvectors, and 𝑓[𝑥ᵢ, 𝑥ᵣ] indicates that the variable 𝑥ᵣ is substituted with 𝑥ᵢ in 𝑓.

5.2.3 Fault-packing. Symbolic evaluation generates a large number of insecurity checks for some leakage models (e.g. memory obliviousness, constant-time). The fault-packing (FP) optimization gathers these insecurity checks along a path and postpones their resolution to the end of the basic block.

Example 3 (Fault-packing). For example, let us consider a basic-block with a path predicate 𝜋. If there are two memory accesses along the basic block that evaluate to ⟨𝜙ᵢ | 𝜙ᵣ⟩ and ⟨𝜙ᵢ | 𝜙ᵣ⟩, we would normally generate two insecurity queries (𝜋 ∧ 𝜙ᵢ # 𝜙ᵣ) and (𝜋 ∧ 𝜙ᵢ # 𝜙ᵣ)—one for each memory access. Fault-packing regroups these checks into a single query (𝜋 ∧ ((𝜙ᵢ # 𝜙ᵣ) ∨ (𝜙ᵢ # 𝜙ᵣ))) 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 simple—they are all evaluated under the same path predicate and are not nested. Therefore, they never end up in a performance degradation (see Table 6).

5.3 Theorems

Theorems and proof sketches are given for the constant-time property. In the companion technical report [49], we detail the full proofs and discuss how the theorems and proofs can be adapted to other leakage models.

⁴We implement untainting with a cache of “untainted variables” that are substituted in the program copy during symbolic evaluation of expressions.
In order to define properties of our symbolic execution, we use $\rightarrow^k$ (resp. $\rightarrow^k$), with $k$ a natural number, to denote $k$ steps in the concrete (resp. symbolic) evaluation.

**Proposition 1.** If a program $P$ is constant-time up to $k$ then for all $i \leq k$, $P$ is constant-time up to $i$.

**Hypothesis 1.** Through this section we assume that theory QF_ABV is correct and complete w.r.t. our concrete evaluation.

The satisfiability problem for the theory QF_ABV is decidable [75]. Therefore we make the following hypothesis on the solver:

**Hypothesis 2.** We suppose that the SMT solver for QF_ABV is correct, complete and always terminates. Therefore for a QF_ABV formula $\phi$, $M \models \pi \iff M \vDash_{\text{SMT}} \pi$.

**Hypothesis 3.** We assume that the program $P$ is defined on all locations computed during the symbolic execution—notably by the function $\text{to\_loc}$ in rule $\text{V\_JUMP}$. Under this hypothesis, and because the solver always terminates (Hypothesis 2), symbolic execution is stuck if and only if a leakage predicate evaluates to false. In this case, an expression $\hat{\phi}$ is leaked such that $\text{secLeak}(\hat{\phi}, \pi)$ evaluates to false and the solver returns a model $M$ such that $M \models \pi \land (\hat{\phi}|_M \neq \hat{\phi}|_{\phi'})$ (from Hypothesis 2).

**Proposition 2.** Concrete semantics is deterministic, c.f. rules of the concrete semantics in Fig. 4.

**Hypothesis 4.** We restrict our analysis to safe programs (e.g. no division by 0, illegal indirect jump, segmentation fault).

Under this hypothesis, concrete execution never gets stuck.

**Definition 4 ($\equiv^M_p$).** We define a concretization relation $\equiv^M_p$ 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 $c \equiv^M_p s$ is the concretization of the $p$-side of the symbolic state $s$ with the model $M$. Let $c \triangleq (l_1, r, m)$ and $s \triangleq (l_2, \rho, \mu, \pi)$. Formally, $c \equiv^M_p s$ holds iff $M \models \pi, l_1 = l_2$ and for all expression $e$, either the symbolic evaluation of $e$ gets stuck or we have $(\rho, \mu, \pi) e \vdash \lnot \hat{\phi} \land (M(\hat{\phi}|_M) = \text{bv} \iff e \vdash \text{bv})$

Notice that because both sides of an initial configuration $s_0$ are low-equivalent, the following proposition holds:

**Proposition 3.** For all concrete configurations $c_0$ and $c'_0$ such that $c_0 \equiv^M_l s_0 \land c'_0 \equiv^M_r s_0$, then $c_0 \equiv^M c'_0$.

The following lemma expresses that when the symbolic evaluation is stuck on a state $s_k$, there exist concrete configurations derived from $s_k$ which produce distinct leakages.

**Lemma 1.** Let $s_k$ be a symbolic configuration obtained after $k$ steps. If $s_k$ is stuck, then there exists a model $M$ such that for each concrete configurations $c_k \equiv^M_l s_k$ and $c'_k \equiv^M_r s_k$, the executions from $c_k$ and $c'_k$ produce distinct leakages.

**Proof overview:** The proof goes by case analysis on the symbolic evaluation of $s_k$. Let $s_k$ be a symbolic configuration that is stuck (i.e., a symbolic leakage predicate evaluates to false with a model $M$), then $s_k$ can be concretized using the model $M$, producing concrete states $c_k$ and $c'_k$ such that $c_k \rightarrow c_{k+1}$ and $c'_k \rightarrow c'_{k+1}$. Finally, because the symbolic leakage model does not over-approximate the concrete leakage, i.e., each symbolic leak corresponds to a concrete leak, we have $t \neq t'$.

The following lemma expresses that when symbolic evaluation does not get stuck up to $k$, then for each pair of concrete executions following the same path up to $k$, there exists a corresponding symbolic execution.
Lemma 2. Let \( s_0 \) be a symbolic initial configuration for a program \( P \) that does not get stuck up to \( k \). For every concrete states \( c_0, c_k, c'_0, c'_k \) and model \( M \) such that \( c_0 \models^M s_0 \land c'_0 \models^M r \), if \( c_0 \mathbin{\xrightarrow{t}^k} c_k \) and \( c'_0 \mathbin{\xrightarrow{r}^k} c'_k \) follow the same path, then there exists a symbolic configuration \( s_k \) and a model \( M' \) such that:

\[
(s_k \models^M' s_k \land c'_k \models^M' s_k)
\]

Proof overview: The proof goes by induction on the number of steps \( k \). For each concrete step \( c_{k-1} \rightarrow c_k \) and \( c'_{k-1} \rightarrow c'_k \), we show that, as long as they follow the same path, there is a symbolic step from \( s_{k-1} \rightarrow s_k' \) that models \( c_k \) and \( c'_k \). This follows from the fact that our symbolic execution does not make under-approximations.

5.3.1 Correctness of RelSE. 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).

Theorem 1 (Correctness of RelSE). For every symbolic configurations \( s_0, s_k \) such that \( s_0 \models^k s_k \) and for every concrete configurations \( c_0, c_k \) and model \( M \), such that \( c_0 \models^M s_0 \) and \( c_k \models^M s_k \), there exists a concrete execution \( c_0 \mathbin{\xrightarrow{t}^k} c_k \).

Proof overview: The proof goes by induction on the number of steps \( k \). For each symbolic step \( s_{k-1} \rightarrow s_k \) and model \( M_k \) such that \( c_{k-1} \models^M s_{k-1} \) and \( c_k \models^M s_k \), there exists a step \( c_{k-1} \rightarrow c_k \) in concrete execution. For each rule, we show that there exists a unique step from \( c_{k-1} \rightarrow s_k' \) (from Proposition 4 and Proposition 2), and, because there is no over-approximation in symbolic execution, \( c_k' \) satisfies \( c_k' \models^M s_k \).

5.3.2 Correct bug-finding for CT. The following theorem expresses that when the symbolic execution gets stuck, then the program is not constant-time.

Theorem 2 (Bug-Finding for CT). Let \( s_0 \) be an initial symbolic configuration for a program \( P \). If symbolic evaluation gets stuck in a configuration \( s_k \) then \( P \) is not constant-time at step \( k \). Formally, if there is a symbolic evaluation \( s_0 \mathbin{\xrightarrow{t}^k} s_k \) such that \( s_k \) is stuck, then there exists a model \( M \) and concrete configurations \( c_0 \models^M s_0, c'_0 \models^M r \), \( c_k \models^M s_k \) and \( c'_k \models^M s_k \) such that:

\[
c_0 \models^L c'_0 \land c_0 \mathbin{\xrightarrow{t}^k} c_k \mathbin{\xrightarrow{r}^k} c_{k+1} \land c'_0 \mathbin{\xrightarrow{t}^k} c'_k \mathbin{\xrightarrow{r}^k} c'_{k+1} \land t_k \neq t'_k
\]

Proof: Let us consider symbolic configurations \( s_0 \) and \( s_k \) such that \( s_0 \mathbin{\xrightarrow{t}^k} s_k \) and \( s_k \) is stuck. From Lemma 1, there is a model \( M \) and concrete configurations \( c_k \) and \( c'_k \) such that \( c_k \models^M s_k \) and \( c'_k \models^M s_k \), and \( c_k \mathbin{\xrightarrow{r}^k} c_{k+1} \) and \( c'_k \mathbin{\xrightarrow{r}^k} c'_{k+1} \) with \( t_k \neq t'_k \). Additionally, let \( c_0, c'_0 \) be concrete configurations such that \( c_0 \models^M s_0 \) and \( c'_0 \models^M s_0 \). From Proposition 3, we have \( c_0 \models^L c'_0 \), and from Theorem 1, there are concrete executions \( c_0 \mathbin{\xrightarrow{t}^k} c_k \) and \( c'_0 \mathbin{\xrightarrow{r}^k} c'_k \). Therefore, we have \( c_0 \mathbin{\xrightarrow{t}^k} c_k \mathbin{\xrightarrow{r}^k} c_{k+1} \) and \( c'_0 \mathbin{\xrightarrow{t}^k} c'_k \mathbin{\xrightarrow{r}^k} c'_{k+1} \) with \( c_0 \models^L c'_0 \) and \( t_k \neq t'_k \), meaning that \( P \) is not constant-time at step \( k \).

5.3.3 Relative completeness of RelSE. 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). Notice that our definition of completeness differs from standard definitions of completeness in SE [37]. Here, completeness up to \( k \) only applies to programs that are constant-time up to \( k \). This directly follows from the fact that our symbolic evaluation blocks on errors while concrete execution continues.
Theorem 3 (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 $c_0 \equiv^M s_0 \land c'_0 \equiv^M s_0$, if $c_0 \xrightarrow{t} c_k$ and $c'_0 \xrightarrow{t'} c'_k$ then there exists a symbolic configuration $s_k$ and a model $M'$ such that:

$$s_0 \xrightarrow{k} s_k \land c_k \equiv^M s_k \land c'_k \equiv^M s_k$$

Proof: First, note that from Theorem 2 and the hypothesis that $P$ is constant-time up to $k$, we know that symbolic evaluation from $s_0$ does not get stuck up to $k$. Knowing this, we can apply Lemma 2 which directly entails Theorem 3. □

5.3.4 Correct bounded-verification for CT. Finally, we prove that if symbolic execution does not get stuck due to a satisfiable insecurity query, then the program is constant-time.

Theorem 4 (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 $k$, $s_0 \xrightarrow{k} s_k$ then for all initial configurations $c_0$ and $c'_0$ and model $M$ such that $c_0 \equiv^M s_0$, and $c'_0 \equiv^M s_0$,

$$c_0 \xrightarrow{t} c_k \land c'_0 \xrightarrow{t'} c'_k \implies t = t'$$

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

Proof overview: The proof goes by induction on the number of steps. If the program is constant-time up to $k - 1$ (induction hypothesis) then from Lemma 2 there is a symbolic execution for any configurations $c_{k-1}$ and $c'_{k-1}$. If these configurations produce distinct leakages, then symbolic execution stuck at step at step $k - 1$ which is a contradiction. This relies on the fact that the symbolic leakage model does not under-approximate the concrete leakage. □

6 EXPERIMENTAL RESULTS

Implementation. We implemented our relational symbolic execution, Binsec/Rel, on top of the binary-level analyzer Binsec [56]. Binsec/Rel takes as input an 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 identified violations together with counterexamples (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 ensures that they are not satisfiable. It can be configured according to the leakage model. We explore the program in a depth-first search manner and we rely on the Boolector SMT-solver [76], currently the best on theory QF_ABV [77, 74].

Research questions. We investigate the following research questions:

RQ1. Effectiveness: constant-time analysis on real-world cryptographic code. 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 with standard approaches. How does Binsec/Rel scale compared to traditional approaches based on self-composition (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 symbolic execution (SE), and can our simplifications be useful for standard SE?

RQ6. Effectiveness: large scale analysis of scrubbing functions. Is Binsec/Rel able to verify the secret-erasure property on a large number of binaries?

Setup. Experiments were performed on a laptop with an Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz processor and 32GB of RAM. Similarly to related work (e.g. [30]), 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. When not stated otherwise, programs are compiled for a x86 (32bit) architecture with their default compiler setup.

Legend. Throughout this section, #I denotes the number of static instructions of a program, #Iunr is the number of unrolled instructions explored by the analysis, P is the number of program paths explored, Time is the execution time give in seconds and • is the number of bugs (vulnerable instructions) found. Status is set to ✓ for secure (exhaustive exploration), ✗ for insecure, or 🕒 for timeout (set to 1 hour). Additionally, for each program, we report the type of operation performed and the length of the secret key (Key) and message (Msg) when applicable (in bytes).

6.1 Effectiveness of Binsec/Rel (RQ1)

We carry out two experiments to assess the effectiveness of our technique: (1) bounded-verification of secure cryptographic primitives previously verified at source- or LLVM-level [21, 22, 13] (Section 6.1.1), (2) automatic replay of known bug studies [15, 22, 50] (Section 6.1.2). 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).

6.1.1 Bounded-Verification. We analyze a large range of secure constant-time cryptographic primitives (296 samples, 64k instructions), comprising:

- Several basic constant-time utility functions such as selection functions [15], sort functions [78] and utility functions from HACL* [13] and OpenSSL [79], compiled with clang (versions 3.0, 3.9 and 7.1), and gcc (versions 5.4 and 8.3) and for optimizations levels O0 and O3;
- A set of representative constant-time cryptographic primitives already studied in the literature on source code [21] or LLVM [22], including implementations of TEA [80], Curve25519-donna [71], aes and des encryption functions taken from BearSSL [81], cryptographic primitives from libodium [12], and the constant-time padding remove function tls-cbc-remove-padding, extracted from OpenSSL [22];
- A set of functions from the HACL* library [13].

Results are reported in Table 2. 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, fixing the input size bounds loops. Additionally, the scalability of Binsec/Rel according to the size of the input data is evaluated in the companion technical report [49] and unbounded loops are discussed in Section 7. 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 ReSE (see Section 6.3). Moreover, this is the first automatic constant-time analysis of these cryptographic libraries at the binary-level.
### Table 2. Bounded verification for constant-time cryptographic implementations.

<table>
<thead>
<tr>
<th>Description and number of binaries</th>
<th>≈ #I</th>
<th>#I_{b,ar}</th>
<th>P</th>
<th>Time</th>
<th>Status</th>
<th>Type</th>
<th>Key</th>
<th>Msg</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Utility</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ct-select (×29)</td>
<td>1015</td>
<td>1507</td>
<td>20</td>
<td>0.2</td>
<td>29 x</td>
<td>-</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>ct-sort (×12)</td>
<td>2400</td>
<td>1782</td>
<td>12</td>
<td>0.2</td>
<td>25 x</td>
<td>-</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>HACL⁺ (×110)</td>
<td>3850</td>
<td>90953</td>
<td>110</td>
<td>7.6</td>
<td>110 x</td>
<td>Utility functions</td>
<td>-</td>
<td>2-200</td>
</tr>
<tr>
<td>OpenSSL (×130)</td>
<td>4550</td>
<td>5113</td>
<td>130</td>
<td>0.9</td>
<td>130 x</td>
<td>-</td>
<td>4-12</td>
<td></td>
</tr>
<tr>
<td><strong>Tea</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decrypt -00</td>
<td>290</td>
<td>953</td>
<td>1</td>
<td>0.1</td>
<td>✓</td>
<td>Block cipher</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>decrypt -03</td>
<td>250</td>
<td>804</td>
<td>1</td>
<td>0.1</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Donna</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-00</td>
<td>7083</td>
<td>10.2M</td>
<td>1</td>
<td>1008.5</td>
<td>✓</td>
<td>Elliptic curve</td>
<td>32</td>
<td>-</td>
</tr>
<tr>
<td>-03</td>
<td>4643</td>
<td>2.7M</td>
<td>1</td>
<td>347.1</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Libsodium</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>salsa20</td>
<td>1627</td>
<td>38.0k</td>
<td>1</td>
<td>3.5</td>
<td>✓</td>
<td>Stream cipher</td>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td>chacha20</td>
<td>2717</td>
<td>12.3k</td>
<td>1</td>
<td>1.5</td>
<td>✓</td>
<td>Stream cipher</td>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td>sha256</td>
<td>4879</td>
<td>48.4k</td>
<td>1</td>
<td>4.5</td>
<td>✓</td>
<td>Secure hash</td>
<td>-</td>
<td>256</td>
</tr>
<tr>
<td>sha512</td>
<td>16312</td>
<td>92.0k</td>
<td>1</td>
<td>8.1</td>
<td>✓</td>
<td>Secure hash</td>
<td>-</td>
<td>256</td>
</tr>
<tr>
<td><strong>Hacl⁺</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chacha20</td>
<td>1221</td>
<td>6.7k</td>
<td>1</td>
<td>4.3</td>
<td>✓</td>
<td>Stream cipher</td>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td>sha256</td>
<td>1279</td>
<td>21.0k</td>
<td>1</td>
<td>2.7</td>
<td>✓</td>
<td>Secure hash</td>
<td>-</td>
<td>256</td>
</tr>
<tr>
<td>sha512</td>
<td>2013</td>
<td>47.5k</td>
<td>1</td>
<td>5.2</td>
<td>✓</td>
<td>Secure hash</td>
<td>-</td>
<td>256</td>
</tr>
<tr>
<td>curve25519</td>
<td>8522</td>
<td>9.4M</td>
<td>1</td>
<td>927.8</td>
<td>✓</td>
<td>Elliptic curve</td>
<td>32</td>
<td>-</td>
</tr>
<tr>
<td><strong>BearSSL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aes-ct-cbcenc⁺</td>
<td>357</td>
<td>3.5k</td>
<td>1</td>
<td>0.5</td>
<td>✓</td>
<td>Block cipher</td>
<td>240</td>
<td>32</td>
</tr>
<tr>
<td>des-ct-cbcenc⁺</td>
<td>682</td>
<td>19.9k</td>
<td>1</td>
<td>12.1</td>
<td>✓</td>
<td>Block cipher</td>
<td>384</td>
<td>16</td>
</tr>
<tr>
<td><strong>OpenSSL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tls-remove-padding-patch</td>
<td>424</td>
<td>35.7k</td>
<td>520</td>
<td>438.0</td>
<td>✓</td>
<td>Remove padding</td>
<td>-</td>
<td>63</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>296 binaries</td>
<td>64114</td>
<td>22.8M</td>
<td>815</td>
<td>2772.7</td>
<td>296 x</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Bounded verification for constant-time cryptographic implementations. † A line in which the number of binaries is not indicated corresponds to 1 binary. ‡ aes set to 2 rounds and des set to 2 iterations.

#### 6.1.2 Bug-Finding

We take three known bug studies from the literature [15, 78, 50] and replay them automatically at binary-level (42 samples, 6k instructions), including: (1) binaries compiled from constant-time sources of a selection function [15] and sort functions [78], (2) non-constant-time versions of aes and des from BearSSL [81], (3) the non-constant-time version of OpenSSL’s tls-cbc-remove-padding⁵ responsible for the famous Lucky13 attack [50].

Results are reported in Table 3 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 prior analysis:

- function ct_select_v1 [15] 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 [22] (LLVM bitcode compiled with clang), are vulnerable when compiled with gcc -O0 or clang -O3 -m32 -march=i386 (details in Section 6.2).

### Conclusion (RQ1)

We perform an extensive analysis over 338 samples of representative cryptographic primitive studied in the literature [21, 13, 22]. Overall, it demonstrates that Binsec/Rel does scale to realistic applications for both bug-finding and bounded-verification. As a side-result, we also proved CT-secure 296 binaries of interest.

⁵https://github.com/openssl/openssl/blob/OpenSSL_1_0_1/ssl/d1_enc.c

Manuscript submitted to ACM
Table 3. Bug-finding for constant-time in cryptographic implementations. † A line in which the number of binaries is not indicated corresponds to 1 binary. ‡ aes set to 2 rounds and des set to 2 iterations.

6.2 Preservation of Constant-Time by Compilers (RQ2).

In this section, we present an easily extensible framework, based on Binsec/Rel, to check constant-time for small programs under multiple compiler setups⁶. Using this framework, we replay a prior manual study [15], which analyzed whether clang optimizations break the constant-time property, for 5 different versions of a selection function (ct-select). We reproduce their analysis in an automatic manner and extend it significantly, adding: 29 new functions, 3 newer version of clang (7.1.0, 9.0.1 and 11.0.1), the gcc compiler, and 2 new architectures (i.e., i686 and arm, while only i386 was considered in the initial study)—for a total of 4148 configurations (192 in the initial study).

Additionally, we investigate the impact of individual optimizations on the preservation of constant-time. For clang, we target the -x86-cmov-converter which converts x86 cmov instructions into branches when profitable and which is known to play a role in the preservation of constant-time [82]. In particular, we evaluate the impact of selectively disabling this optimization, by passing the flags -O3 -mllvm -x86-cmov-converter=0 to clang, which we denote O3⁻. For gcc, we target the if-conversion (i.e., -fif-conversion -fif-conversion2 -ftree-loop-if-convert), which transforms conditional jumps into branchless equivalent. In particular, we evaluate the impact of selectively enabling this optimization, by passing the flags -O0 -fif-conversion -fif-conversion2 -ftree-loop-if-convert to gcc, (denoted O0⁺); and the impact of selectively disabling this optimization using -O3 -fno-if-conversion -fno-if-conversion2 -fno-tree-loop-if-convert (denoted O3⁻). Bear in mind that the i386 architecture does not feature cmov instructions but i686 does. Results are presented in Table 4. Results for O3⁻ are not applicable to clang-3.0 and clang-3.9 (denoted - in the table) as these versions do not recognize the -x86-cmov-converter argument.

We confirm the main conclusion of Simon et al. [15] that clang is more likely to optimize away constant-time protections as the optimization level increases. However, contrary to their work, our experiments show that newer versions of clang are not necessarily more likely than older ones to break constant-time (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. Indeed, the compiler takes advantage of the many ARM conditional instructions to remove conditional jumps in sort_naive and naive_select. This also applies to the i686 architecture but only for naive_select. We conclude that the if-conversion passes of gcc play a role here, as disabling them (O3⁻) produces insecure binaries. However, the

⁶https://github.com/binsec/rel_bench/tree/main/properties_vs_compilers/ct

Manuscript submitted to ACM
We also remark that disabling the `-x86-cmov-converter` versions and options of gcc shows that insecure binaries for two different reasons (details are provided in the technical report [49]).

Conclusion (RQ2). This study shows that Binsec/Rel is generic in the sense that it can be applied with different versions and options of clang and gcc, over x86 and ARM. We also get the following interesting results:

- We found that, contrary to clang, gcc optimizations tend to help enforcing constant-time—gcc `-O3` preserves constant-time in all our examples. gcc even sometimes produces secure binaries from insecure sources thanks to the if-conversion passes;
- We found that backend passes of clang can introduce vulnerabilities in codes that are secure at the LLVM level;
- We found that clang use of cmov instructions might introduce secret-dependent memory accesses;
- Finally, this study shows that the preservation of constant-time by compilers depends on multiple factors and cannot simply rely on enabling/disabling optimizations. Instead, compiler-based hardening [83, 84] or property preservation [15] seem promising directions, in which Binsec/Rel could be used for validation.

---

We did confirm that `ct-verif` with the setting `-clang-options=-O3 -m32 -march=i386` does not report the vulnerability.

Table 4. Preservation of constant-time for several programs compiled with gcc or clang for i386, i686 or arm architectures and optimization levels 0, O1, 02, 03 or O3. ✓ indicates that a program is secure whereas c (resp. m) indicates that a program is insecure due to secret-dependent control-flow (resp. memory access). gcc-all denotes versions 5.4.0, 6.2.0, 7.2.0, 8.3.0 and 10.2.0 of gcc.

<table>
<thead>
<tr>
<th>compiler arch opt-level</th>
<th>clang-3.0 i386/i686</th>
<th>clang-3.9 i386/i686</th>
<th>clang-7.1 i386</th>
<th>clang-7.1 i686</th>
<th>clang-9/11 i386</th>
<th>clang-9/11 i686</th>
<th>gcc-all i386</th>
<th>gcc-all i686</th>
<th>gcc-10.3 arm</th>
</tr>
</thead>
<tbody>
<tr>
<td>ct_select_v1</td>
<td>✓✓ ✓ ✓ c c -</td>
<td>✓✓ ✓ ✓ c c -</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ct_select_v2</td>
<td>✓✓ ✓ ✓ c c -</td>
<td>✓✓ ✓ ✓ c c -</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ct_select_v3</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ct_select_v4</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>naive_select</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sort</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sort_multiplex</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sort_naive</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OpenSSL-utility x11</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HACL*-utility x11</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>tea-enc/dec x2</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓✓ ✓ ✓ ✓✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Fact that O0 is still insecure shows that if-conversion passes must be combined with other optimizations (at least O1) to effectively remove conditional jumps.

Finally, we found that constant-time sort functions, taken from the benchmark of the Binsec/Rel tool, can be compiled to insecure binaries for two different reasons (details are provided in the technical report [49]).

- For the i386 architecture and old compilers, conditional select LLVM instructions are compiled to conditional jumps because target architectures do not feature cmov instructions. These violations are introduced in backend passes of clang, making them of reach of LLVM verification tools like ct-verif;
- More interestingly, we found that for more recent architectures featuring cmov (i.e., i686), the use of cmov might introduce secret-dependent memory accesses. Indeed, the compiler introduces a secret-dependent pointer selection, done with cmov, which results in a memory-based leak when the pointer is dereferenced.

We also remark that disabling the `-x86-cmov-converter` does not change anything in our settings.
6.3 Comparison against Standard Techniques (RQ3,RQ4,RQ5)

We compare Binsec/Rel with standard techniques based on self-composition and relational symbolic execution (RelSE) (Section 6.3.1), then we analyze the performance of our different simplifications (Section 6.3.2), and finally we investigate the overhead of Binsec/Rel compared to standard SE, and whether our simplifications are useful for SE (Section 6.3.3).

Experiments are performed on the programs introduced in Section 6.1 for bug-finding and bounded-verification (338 samples, 70k instructions). We report the following metrics: total number of unrolled instruction \(\#I_{\text{unr}}\), number of instruction explored per seconds \(\#I_{\text{unr}}/s\), 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.

### 6.3.1 Comparison vs. Standard Approaches (RQ3).

We evaluate Binsec/Rel against SC and RelSE. Since no implementation of these methods fits our particular use-cases, we implement them directly in Binsec. RelSE is obtained by disabling Binsec/Rel optimizations (Section 5.2), 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 5.

<table>
<thead>
<tr>
<th></th>
<th>#I_{\text{unr}}</th>
<th>#I_{\text{unr}}/s</th>
<th>#Q</th>
<th>#Q_e</th>
<th>#Q_i</th>
<th>Time (s)</th>
<th>✓</th>
<th>✗</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC</td>
<td>248k</td>
<td>3.9</td>
<td>158k</td>
<td>14k</td>
<td>143k</td>
<td>64296</td>
<td>16</td>
<td>280</td>
<td>42</td>
</tr>
<tr>
<td>RelSE</td>
<td>349k</td>
<td>6.2</td>
<td>90k</td>
<td>17k</td>
<td>73k</td>
<td>56428</td>
<td>13</td>
<td>283</td>
<td>42</td>
</tr>
<tr>
<td>Binsec/Rel</td>
<td>22.8M</td>
<td>6238</td>
<td>3700</td>
<td>1292</td>
<td>0</td>
<td>3657</td>
<td>0</td>
<td>296</td>
<td>42</td>
</tr>
</tbody>
</table>

Table 5: Binsec/Rel vs. standard approaches.

While RelSE performs slightly better than SC (1.6× speedup in terms of \#I_{\text{unr}}/s) thanks to a noticeable reduction of the number of queries (approximately 50%), both techniques are not efficient enough on binary code: RelSE times out in 13 cases and achieves an analysis speed of only 6.2 instructions per second while SC is worse. Binsec/Rel completely outperforms both previous approaches:

- The optimizations implemented in Binsec/Rel drastically reduce the number of queries sent to the solver (57× less insecurity queries than RelSE);
- Binsec/Rel reports no timeout, is 1000× faster than RelSE and 1600× faster than SC in terms of \#I_{\text{unr}}/s;
- Binsec/Rel can perform bounded-verification of large programs (e.g. donna, des-ct, chacha20, etc.) that were out of reach of prior approaches.

### 6.3.2 Performance of Simplifications (RQ4).

We evaluate the performance of our individual optimizations: on-the-fly read-over-write (FlyRow), untainting (Unt) and fault-packing (FP). Results are reported in Table 6:

- FlyRow is the major source of improvement in Binsec/Rel, drastically reducing the number of queries sent to the solver and allowing a 718× speedup compared to RelSE in terms of \#I_{\text{unr}}/s;
- Untainting and fault-packing do have a positive impact on RelSE—untainting alone reduces the number of queries by almost 50%, the two optimizations together yield a 2× speedup;
- Yet, their impact is more modest once FlyRow is activated: untainting leads to a very slight slowdown, while fault-packing achieves a 1.4× 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 (almost 11× faster).

6.3.3 Comparison vs. Standard SE (RQ5). We investigate the overhead of Binsec/Rel compared to standard symbolic execution (SE); evaluate whether on-the-fly read-over-write (FlyRow) can improve performance of SE; and also compare FlyRow to a recent implementation of read-over-write [74] (PostRow), implemented posterior to symbolic-execution as a formula pre-processing. Standard symbolic-execution is directly implemented in the Rel module and models a single execution of the program with exploration queries but without insecurity queries.

Table 6. Performances of Binsec/Rel simplifications.

<table>
<thead>
<tr>
<th>Version</th>
<th>#\text{l\text{unr}}</th>
<th>#\text{l\text{unr}}/s</th>
<th>#Q</th>
<th>#Q_{e}</th>
<th>#Q_{i}</th>
<th>Time</th>
<th>\text{\xmark}</th>
<th>\text{\checkmark}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard RelSE</td>
<td>RelSE</td>
<td>349k</td>
<td>6.2</td>
<td>90148</td>
<td>17428</td>
<td>72720</td>
<td>56429</td>
<td>13</td>
</tr>
<tr>
<td>(without FlyRow)</td>
<td>+ Unt</td>
<td>414k</td>
<td>9.9</td>
<td>48648</td>
<td>20601</td>
<td>28047</td>
<td>41852</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>+ FP</td>
<td>437k</td>
<td>12.7</td>
<td>35100</td>
<td>21834</td>
<td>13266</td>
<td>34471</td>
<td>7</td>
</tr>
<tr>
<td>Binsec/Rel</td>
<td>RelSE + FlyRow</td>
<td>22.8M</td>
<td>4450</td>
<td>3738</td>
<td>2408</td>
<td>1330</td>
<td>5127</td>
<td>0</td>
</tr>
<tr>
<td>(with FlyRow)</td>
<td>+ Unt</td>
<td>22.8M</td>
<td>4429</td>
<td>3738</td>
<td>2408</td>
<td>1330</td>
<td>5151</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>+ FP</td>
<td>22.8M</td>
<td>6238</td>
<td>3700</td>
<td>2408</td>
<td>1292</td>
<td>3658</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7. Performances of RelSE compared to standard SE with/without binary level simplifications.

<table>
<thead>
<tr>
<th>Version</th>
<th>#\text{l\text{unr}}</th>
<th>#\text{l\text{unr}}/s</th>
<th>#Q</th>
<th>Time</th>
<th>Version</th>
<th>#\text{l\text{unr}}</th>
<th>#\text{l\text{unr}}/s</th>
<th>#Q</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE</td>
<td>522k</td>
<td>19.5</td>
<td>24444</td>
<td>26814</td>
<td>RelSE</td>
<td>349k</td>
<td>6.2</td>
<td>90148</td>
<td>56429</td>
</tr>
<tr>
<td>SE+PostRow [74]</td>
<td>628k</td>
<td>29.2</td>
<td>29389</td>
<td>21475</td>
<td>RelSE+PostRow</td>
<td>317k</td>
<td>5.3</td>
<td>65834</td>
<td>60295</td>
</tr>
<tr>
<td>SE+FlyRow</td>
<td>22.8M</td>
<td>12531.1</td>
<td>534</td>
<td>1817</td>
<td>Binsec/Rel</td>
<td>22.8M</td>
<td>6237.7</td>
<td>3700</td>
<td>3657</td>
</tr>
</tbody>
</table>

- Binsec/Rel, compared to our best setting for symbolic execution (SE+FlyRow), only has an overhead of 2x in terms of #\text{l\text{unr}}/s. Hence constant-time comes with an acceptable overhead on top of standard symbolic execution. This is consistent with the fact that our simplifications discard most insecurity queries, letting only the exploration queries which are also part of symbolic-execution;
- For RelSE, FlyRow completely outperforms PostRow. First, PostRow is not designed for relational verification and duplicates the memory. Second, PostRow simplifications are not propagated along the execution and must be recomputed for every query, producing a significant simplification 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 643 in our experiments, performing much better than PostRow (430× faster).

Conclusion (RQ3, RQ4, RQ5). Binsec/Rel performs significantly better than previous approaches to relational symbolic execution (1000× speedup vs. RelSE). The main source of improvement is the on-the-fly read-over-write simplification (FlyRow), which yields a 718× speedup vs. RelSE and sends 57× less insecurity queries to the solver. Note that, in our context, FlyRow 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 on insecure programs. Finally, in our experiments, FlyRow significantly improves performance of standard symbolic-execution (643× speedup).

6.4 Preservation of Secret-Erasure by Compilers (RQ6)

Secret-erasure is usually enforced using scrubbing functions—functions that overwrite a given part of the memory with dummy values. In this section we present a framework to automatically check the preservation of secret-erasure for multiple scrubbing functions and compilers. This framework is open source\(^8\) and can be easily extended with new compilers and new scrubbing functions. Using Binsec/Rel, we analyze 17 scrubbing functions; with multiple versions of clang (3.0, 3.9, 7.1.0, 9.0.1 and 11.0.1) and gcc (5.4.0, 6.2.0, 7.2.0, 8.3.0 and 10.2.0); and multiple optimization levels (−O0, −O1, −O2 and −O3). We also investigate the impact of disabling individual optimizations (those related to the dead-store-elimination pass) on the preservation of secret-erasure (cf. Section 6.4.7). This accounts for a total of 1156 binaries and extends a prior manual study on scrubbing mechanisms [18].

In this section, clang-all-versions (resp. gcc-all-versions) refer to all the aforementioned clang (resp. gcc) versions; and in tables ✓ indicates that a program is secure and ✗ that it is insecure w.r.t secret-erasure.

6.4.1 Naive implementations. First, we consider naive (insecure) implementations of scrubbing functions:

- Loop: naive scrubbing function that uses a simple for loop to set the memory to 0,
- memset: uses the memset function from the Standard C Library,
- bzero: function defined in glibc to set memory to 0.

<table>
<thead>
<tr>
<th></th>
<th>clang-all-versions</th>
<th>gcc-5.4/6.2/7.2</th>
<th>gcc-8.3/10.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>−O0 −O1 −O2 −O3</td>
<td>−O0 −O1 −O2 −O3</td>
<td>−O0 −O1 −O2 −O3</td>
</tr>
<tr>
<td>loop</td>
<td>✓ ✓ X X</td>
<td>✓ ✓ X X</td>
<td>✓ ✓ X X</td>
</tr>
<tr>
<td>memset</td>
<td>✓ ✓ X X</td>
<td>✓ ✓ X X</td>
<td>✓ ✓ X X</td>
</tr>
<tr>
<td>bzero</td>
<td>✓ ✓ X X</td>
<td>✓ ✓ X X</td>
<td>✓ ✓ ✓ X</td>
</tr>
</tbody>
</table>

Table 8. Preservation of secret-erasure for naive scrubbing functions.

Results (cf. Table 8). As expected, without appropriate countermeasures, these naive implementation of scrubbing functions are all optimized away by all versions of clang and gcc at optimization level −O2 and −O3. Additionally, as highlighted in Table 8, bzero is also optimized away at optimization level −O1 with gcc−7.2.0 and older versions\(^9\).

6.4.2 Volatile function pointer. The volatile type qualifier indicates that the value of an object may change at any time, preventing the compiler from optimizing memory accesses to volatile objects. This mechanism can be exploited for secure secret-erasure by using a volatile function pointer for the scrubbing function (e.g. eventually redirecting to memset). Because the function may change, the compiler cannot optimize it away. Listing 3 illustrates the implementation of this mechanism in OpenSSL [85].

\(^8\)https://github.com/binsec/rel_bench/tree/main/properties_vs_compilers/secret-erasure
\(^9\)This is because the function calls to scrub and bzero are inlined in gcc−7.2.0 and older versions, making the optimization possible whereas the call to scrub is not inlined in gcc−8.3.0 and older versions.

Manuscript submitted to ACM
typedef void *(*memset_t)(void *, int, size_t);
static volatile memset_t memset_func = memset;
void scrub(char *buf, size_t size) {
    memset_func(buf, 0, size);
}

Listing 3. OpenSSL scrubbing function [85]. Relies on volatile function pointer.

Results (cf. Table 9). Binsec/Rel reports that, for all versions of gcc, the secret-erasure property is not preserved at optimization levels -O2 and -O3. Indeed, the caller-saved register edx is pushed on the stack before the call to the volatile function. However, it contains secret data which are spilled on the stack and not cleared afterwards. This shows that our tool can find violations of secret erasure from register spilling. We conclude that while the volatile function pointer mechanism is effective for preventing the scrubbing function to be optimized away, it may also introduce unnecessary register spilling that might break secret-erasure.

6.4.3 Volatile data pointer. The volatile type qualifier can also be used for secure secret-erasure by marking the data to scrub as volatile before erasing it. We analyze several implementations based on this mechanism:

- **ptr_to_volatile_loop** casts the pointer buf to a pointer-to-volatile vbuf (cf. Listing 4, line 1) before scrubbing data from vbuf using a simple for or while loop. This is a commonly used technique for scrubbing memory, used for instance in Libgcrypt [86], wolfSSL [87], or sudo [88];
- **ptr_to_volatile_memset** is similar to ptr_to_volatile_loop but scrubs data from memory using memset. Note that this implementation is insecure as the volatile type qualifier is discarded by the function call—volatile char * is not compatible with void *;
- **volatile_ptr_loop** (resp. volatile_ptr_memset) casts the pointer buf to a volatile pointer vbuf— but pointing to non volatile data (cf. Listing 4, line 2) before scrubbing data from vbuf using a simple for or while loop (resp. memset)10;
- **vol_ptr_to_vol_loop** casts the pointer buf to a volatile pointer-to-volatile vbuf (cf. Listing 4, line 3) before scrubbing data from vbuf using a simple for or while loop. It is the fallback scrubbing mechanism used in libsodium [89] and in HACL* [90] cryptographic libraries;
- **vol_ptr_to_vol_memset** is similar to vol_ptr_to_vol_loop but uses memset instead of a loop.

```
volatile char *vbuf = (volatile char *) buf; // Pointer-to-volatile
char * volatile vbuf = (char *volatile) buf; // Volatile ptr (pointer is volatile itself)
volatile char *volatile vbuf = (volatile char *volatile) buf; // Volatile ptr-to-volatile
```

Listing 4. Different usage of volatile type qualifier before scrubbing memory.

Results (cf. Table 10). First, our experiments show that using volatile pointers to non-volatile data does not reliably prevent the compiler from optimizing away the scrubbing function. Indeed, gcc optimizes away the scrubbing function at optimization level -O2 and -O3 in both volatile_ptr implementations. Second, using a pointer to volatile works

---

10Although we did not find this implementation in real-world cryptographic code, we were curious about how the compiler would handle this case.
in the loop version (i.e., `ptr_to_volatile_loop` and `vol_ptr_to_vol_loop`) but not in the `memset` versions (i.e., `ptr_to_volatile_memset` and `vol_ptr_to_vol_memset`) as the function call to `memset` discards the volatile qualifier.

### 6.4.4 Memory barriers

Memory barriers are inline assembly statements which indicate the compiler that the memory could be read or written, forcing the compiler to preserve preceding store operations. We study four different implementations of memory barriers: three implementations from `safeclib` [91], plus the approach recommended in a prior study on scrubbing mechanisms [18].

- **memory_barrier_simple** (cf. Listing 5, line 1) is the implementation used in `explicit_bzero` and the fallback implementation used in `safeclib`. As pointed by Yang, Johannesmeyer, Olesen, Lerner, and Levchenko [18], this barrier works with `gcc` [92] but might not work with `clang`, which might optimize away a call to `memset` or a loop before this barrier [93]—although we could not reproduce the behavior in our experiments;
- **memory_barrier_mfence** (cf. Listing 5, line 2) is similar to `memory_barrier_simple` with an additional `mfence` instruction for serializing memory. It is used in `safeclib` when `mfence` instruction is available;
- **memory_barrier_lock** (cf. Listing 5, line 3) is similar to `memory_barrier_mfence` but uses a `lock` prefix for serializing memory. It is used in `safeclib` on `i386` architectures;
- **memory_barrier_ptr** (cf. Listing 5, line 4) is a more resilient approach than `memory_barrier_simple`, recommended in the study of Yang, Johannesmeyer, Olesen, Lerner, and Levchenko [18], and used for instance in `libsodium_memzero` [89]. It makes the pointer `buf` visible to the assembly code, preventing prior store operation to this pointer from being optimized away.

Listing 5. Different implementation of memory barriers.

<table>
<thead>
<tr>
<th></th>
<th>clang-all-versions</th>
<th>gcc-all-versions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-00 -01 -02 -03</td>
<td>-00 -01 -02 -03</td>
</tr>
<tr>
<td><code>ptr_to_volatile_loop</code></td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td><code>volatile_ptr_loop</code></td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✗ ✗</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✗ ✗</td>
</tr>
<tr>
<td><code>vol_ptr_to_vol_loop</code></td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

Table 10. Preservation of secret-erasure with volatile data pointers. ✓ indicates that a program is secure and ✗ that it is insecure.

### 6.4.5 Weak symbols

Weak symbols are specially annotated symbols (with `__attribute__((weak))`) whose definition may change at link time. An illustration of a weak function symbol is given in Listing 6. The compiler cannot optimize

```
1 __asm__ __volatile__("":"memory"); // memory_barrier_simple
2 __asm__ __volatile__("mfence" ::: "memory"); // memory_barrier_mfence
3 __asm__ __volatile__("lock; addl $0,0(%%esp)" ::: "memory"); // memory_barrier_lock
4 __asm__ __volatile__(""::"r"(buf):"memory"); // memory_barrier_ptr
```

Listing 6. Different implementation of memory barriers.

**Results.** For all the implementation of memory barriers that we tested, we did not find any vulnerability—even with the version deemed insecure in prior study [18].

### 6.4.5 Weak symbols

Weak symbols are specially annotated symbols (with `__attribute__((weak))`) whose definition may change at link time. An illustration of a weak function symbol is given in Listing 6. The compiler cannot optimize

11 As explained in a bug report [93], `memory_barrier_simple` is not reliable because `clang` might consider that the inlined assembly code does not access the buffer (e.g. by fitting all of the buffer in registers). The fact that we were not able to reproduce this bug in our setup is due to differences in programs (in our program the address of the buffer escapes because of function calls whereas it is not the case in the bug report); it does not mean that this barrier is secure (it is not).
a store operation preceding the call to _sodium_dummy_symbol because its definition may change and could access the content of the buffer. This mechanism is used in libsnodum memzero [89] when weak symbols are available.

```c
__attribute__((weak)) void _sodium_dummy_symbol(void *const pnt, const size_t len) {
    (void) pnt; (void) len;
}

void scrub(char *buf, size_t size) {
    memset(buf, 0, size);
    _sodium_dummy_symbol(buf, size);
}
```

Listing 6. Libsodium implementation of weak symbols for memory scrubbing.

**Results.** Binsec/Rel did not find any vulnerability with weak-symbols.

6.4.6 Off-the-shelf implementations. Finally, we consider two secure implementations of scrubbing functions proposed in external libraries, namely explicit_bzero and memset_s. explicit_bzero is a function defined in glibc to set memory to 0, with additional protections to not be optimized away by the compiler. Similarly, memset_s is a function defined in the optional Annex K (bound-checking interfaces) of the C11 standard, which sets a memory region to a given value and should not be optimized away. We take the implementation of safeclib [94], compiled with its default Makefile for an i386 architecture. Both implementations both rely on a memory barrier (see Section 6.4.4) to prevent the compiler from optimizing scrubbing operations.

**Results.** Binsec/Rel did not find any vulnerability with these functions.

6.4.7 Impact of disabling individual optimizations. In order to understand what causes compilers to introduce violations of secret-erasure, we selectively disable the -dse (i.e., dead store elimination) option in clang and the -dse and -tree-dse (i.e., dead store elimination on tree) in gcc.

**Results.** For clang-3.9, clang-7.1.0, clang-9.0.1 and clang-11.0.112, disabling the -dse transform pass makes all our samples secure. This points towards the hypothesis that the -dse transform pass is often responsible for breaking secret-erasure and that, in some cases, disabling it might be sufficient to preserve secret-erasure13.

The results for gcc are given in table Table 11. Firstly, we observe that both -dse and -tree-dse play a role in the preservation of secret-erasure. Indeed, for bzero, disabling dse is sufficient for obtaining a secure binary, while for volatile_ptr_memset and vol_ptr_to_vol_memset, -tree-dse must be disabled. On the contrary, for memset and ptr_to_volatile_memset, it is necessary to disable both optimizations. Secondly, we observe that there are other factors that affect the preservation of secret-erasure. Indeed, the volatile_func program is still insecure because of register spilling. Additionally, loop and volatile_ptr_loop are also insecure because the loop is still optimized away.

---

12Clang-3.0 is omitted in this study because we were not able to run the LLVM optimizer (opt) for clang-3.0 in order to disable the -dse optimization.
13However, we strongly suspect that this conclusion does not generalize to all programs, for instance to programs that violate secret-erasure because of register spilling.
Table 11. Preservation of secret-erasure for several scrubbing functions compiled by gcc, with selective optimization disabling. −O3 is the baseline optimization level, dse (resp. tdse) indicates that the −fno-dse (resp. −fno-tree-dse) arguments have been passed to gcc to disable dse (resp. tree-dse) optimization, and “both” indicates that both optimizations have been disabled. Circles highlight the least set of options that must be disabled to make the program secure and bold highlights programs that are insecure in all our configurations.

<table>
<thead>
<tr>
<th>Function</th>
<th>gcc-5.4.0 6.2.0 7.2.0</th>
<th>gcc-8.3.0 10.2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>−O3 dse tdse both</td>
<td>−O3 dse tdse both</td>
</tr>
<tr>
<td>loop</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>memset</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>bzero</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>explicit_bzero</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>volatile_func</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>ptr_to_volatile_loop</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>ptr_to_volatile_memset</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>volatile_ptr_loop</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>volatile_ptr_memset</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>vol_ptr_to_vol_loop</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>vol_ptr_to_vol_memset</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>memory_barrier_all (x4)</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>weak_symbols</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

7 DISCUSSION

Limitations of the technique. The relational symbolic execution introduced in this paper handles loops and recursion with unrolling. Unrolling still enables exhaustive exploration for programs without unbounded loops such as tea or donna. However, for programs with unbounded loops, such as stream ciphers salsa20 or chacha20 it leads to unexplored program paths, and hence might miss violations. A possible solution to enable sound analysis for program with unbounded loops would be to use relational loop invariants however, it would sacrifice bug-finding. Similarly, indirect jump targets are only enumerated up to a given bound, which might lead to unexplored program paths and consequently missed violations. However, we did not encounter incomplete enumerations in our experiments: in the cryptographic primitives that we analyzed indirect jumps had a single (or few) target. Finally, any register or part of the memory that is concretized in the initial state of the symbolic execution might lead to unexplored program behaviors and missed violations. In Binsec/Rel, memory and register are symbolic by default and any concretization (e.g. setting the initial value of esp, or which memory addresses are initialized from the binary) must be made explicitly by the user.

The definition of secret-erasure used in this paper is conservative in the sense that it forbids secret-dependent branches (and hence related implicit flows). We leave for future work the exploration of alternative (less conservative) definitions that could either declassify secret-dependent conditions, or allow secret-secret dependent conditions as long as both branches produce the same observations. Finally, Binsec/Rel restricts to a sequential semantics and hence cannot detect Spectre vulnerabilities however the technique has recently been adapted to a speculative semantics.

Implementation limitations. The implementation of Binsec/Rel shows limitations commonly found in research prototypes: it does not support dynamic libraries (binaries must be statically linked or stubs must be provided for external function calls), it does not support dynamic memory allocation (data structures must be statically allocated), it does not implement predefined system call stubs, it does not support multi-threading, and it does not support floating

\[14\text{In our experiments we fix the input size for these programs, but we could also keep it symbolic and restrict it to a given range, which would extend security guarantees for all input sizes in this range.} \]

\[15\text{Binsec/Rel detects and records incomplete jump target enumerations and, if it cannot find any vulnerabilities, it returns “unknown” instead of “secure”.} \]
point instructions. These problems are orthogonal to the core contribution of this paper. 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 [56], whose efficiency and robustness have been demonstrated in prior large scale studies on both adversarial code and managed code [62, 96, 97, 98]. The IR lifting part has been positively evaluated in an external study [99] and the symbolic engine features aggressive formula optimizations [74]. All our experiments use the same search heuristics (depth-first) and, for bounded-verification, smarter heuristics do not change the performance. Regarding the solver, we also tried Z3 [70] 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 it allows us to compare very close implementations.

8 RELATED WORK

Related work has already been lengthily 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 analysis (Table 12) partly taken from [22].

<table>
<thead>
<tr>
<th>Tool</th>
<th>Target</th>
<th>NI</th>
<th>Technique</th>
<th>P</th>
<th>BV</th>
<th>BF</th>
<th>≈XP max</th>
<th>I_loops/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelSym [48]</td>
<td>imp-for</td>
<td>✓</td>
<td>RelSE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10 LoC</td>
<td>NA</td>
</tr>
<tr>
<td>IF-exploit [40]</td>
<td>Java</td>
<td>✓</td>
<td>self-composition</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>20 LoC</td>
<td>NA</td>
</tr>
<tr>
<td>Type-SC-SE [41]</td>
<td>C</td>
<td>✓</td>
<td>type-based self-comp. + DSE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>20 LoC</td>
<td>NA</td>
</tr>
<tr>
<td>Casym [23]</td>
<td>LLVM</td>
<td>✓</td>
<td>self-comp. + over-approx</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>200 LoC (C)</td>
<td>NA</td>
</tr>
<tr>
<td>ENCoVer [100]</td>
<td>Java bytecode</td>
<td>✓</td>
<td>epistemic logic + DSE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>33k I_u</td>
<td>186</td>
</tr>
<tr>
<td>IF-low-level [39]</td>
<td>binary</td>
<td>✓</td>
<td>self-comp. + invariants</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>250 I_u</td>
<td>NA</td>
</tr>
<tr>
<td>IF-firmware [42]</td>
<td>binary</td>
<td>×</td>
<td>self-comp. + concretizations</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>500k I_u</td>
<td>260</td>
</tr>
<tr>
<td>CacheD [27]</td>
<td>binary</td>
<td>×</td>
<td>tainting + concretizations</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>31M I_u</td>
<td>2010</td>
</tr>
<tr>
<td>Binsec/Rel</td>
<td>binary</td>
<td>×</td>
<td>RelSE + formula simplifications</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>10M I_u</td>
<td>3861</td>
</tr>
</tbody>
</table>

Table 12. Comparison of SE-based tools for information flow analysis. NI indicates whether the tool handles general non-interference (diverging paths) or not. In the column “Technique”, DSE stands for dynamic symbolic execution. P stands for Proof, BV for Bounded-Verification, BF for Bug-Finding. ≈XP max indicates the approximate size of the use cases where LoC stands for Lines of Code, I_u for static instructions, and I_loops for unrolled instructions. Finally, NA indicates Non-Applicable.

**Self-composition and SE** has first been used by Milushev et al. [41]. 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 programs. 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, 39] suffer from the same problems.

*Instead of considering the general case of noninterference, we focus on properties that relate traces following the same path, and we show that it remains tractable for SE with adequate optimizations.*

**Relational symbolic execution.** Shadow symbolic execution [47, 101] aims at efficiently testing evolving software by focusing on the new behaviors introduced by a patch. It introduces the idea of *sharing formulas* across two executions.
in the same SE instance. The term \textit{relational symbolic execution} has been coined more recently \cite{48} but this work is limited to a simple toy imperative language and do not address scalability.

\textit{We maximize sharing between pairs of executions, as ShadowSE does, but we also develop specific optimizations tailored to the case of information-flow analysis at binary-level. Experiments show that our optimizations are crucial in this context.}

\textbf{Scaling SE for information flow analysis.} Only three previous works in this category achieve scalability, yet at the cost of either precision or soundness. Wang et al. \cite{27} and Subramanyan et al. \cite{42} sacrifice soundness for scalability (no bounded-verification). The former performs symbolic execution on fully concrete traces and only symbolizes secrets. The latter concretizes memory accesses. In both cases, they may miss feasible paths as well as vulnerabilities. Brotzman \textit{et al.} \cite{23} 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.

\textit{We adopt a different approach and scale by heavy formula optimizations, allowing us to keep both correct bug-finding and correct bounded-verification. Interestingly, our method is faster than these approximated ones. Moreover, our technique is compatible with the previous approximations for extra-scaling.}

\textbf{Other methods for constant-time analysis.} 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 \cite{102}, dynamic binary instrumentation \cite{25, 28}, dynamic symbolic execution (DSE) \cite{26}, or fuzzing \cite{103}.

\textbf{Static approaches} based on sound static analysis \cite{104, 105, 10, 20, 21, 29, 30, 31, 22, 106} give formal guarantees that a program is free from timing-side-channels but they cannot find bugs when a program is rejected.

Aside from a posteriori analysis, correct-by-design approaches \cite{107, 108, 109, 13} require to reimplement cryptographic primitives from scratch. Program transformations have been proposed to automatically transform insecure programs into (variations of) constant-time programs \cite{110, 104, 105, 111, 112, 23, 113, 84, 112, 83, 114}. In particular, Raccoon and Constantine consider a constant-time leakage model and seem promising, however they operate at LLVM level and do not protect against violations introduced by backend compiler passes. Therefore, Binsec/Rel is complementary to these techniques, as it can be used for investigating code patterns and backend optimizations that may introduce constant-time violations in backend compiler passes.

\textbf{Other methods for secret-erasure.} Compiler or OS-based secure deallocation \cite{115, 116} have been proposed but require compiler or OS-support, in contrast this work focuses on application-based secret-erasure.

Chong and Myers \cite{14} introduce the first framework to specify erasure policies which has been later refined to express richer policies using a knowledge-based approach \cite{117}, and cryptographic data deletion \cite{118}. These works focus on expressing complex secret-erasure policies, but are not directly applicable to concrete languages. Hansen and Probst \cite{119} propose the first application of a simple secret-erasure policy for a concrete language (i.e., Java Card Bytecode), which ensures that secrets are unavailable after program termination. Our definition of secret erasure is close to theirs and directly applicable for binary-level verification.

Most enforcement mechanisms for erasure are language-based and rely on type systems to enforce information flow control \cite{120, 121, 118, 122, 123}. Secretgrind \cite{32}, a dynamic taint tracking tool based on Valgrind \cite{124} to track secret data in memory, is the closest work to ours, with the main difference being that it uses dynamic analysis and permits implicit flows, while we use static analysis and forbid implicit flows.

The problem of (non-)preservation of secret-erasure by compilers is well known \cite{19, 16, 17, 18, 15}. To remedy it, a notion of information flow-preserving program transformation has been proposed \cite{17} but this approach requires
to compile programs using CompCert [125] and does not apply to already compiled binaries. Finally, preservation of erasure functions by compilers has been studied manually [18], and we further this line of work by proposing an extensible framework for automating the process.

9 CONCLUSION

We tackle the problem of designing an automatic and efficient binary-level analyzer for information flow properties, 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 constant-time analysis and secret-erasure for a wide range of cryptographic implementations, and to automate prior manual studies on the preservation of constant-time and secret-erasure by compilers. We highlight incorrect usages of volatile data pointer for secret erasure, and show that scrubbing mechanisms based on volatile function pointers can introduce additional violation from register spilling. We also found three constant-time vulnerabilities that slipped through prior manual and automated analyses, and we discovered that gcc -O0 and backend passes of clang introduce violations of constant-time out of reach of state-of-the-art constant-time verification tools at LLVM or source level.

ACKNOWLEDGMENTS

We would like to thank Guillaume Girol for his help with setting up Nix virtual environments, which enable reproducible compilation in our frameworks, as well as Frédéric Recoules for his help with the final release of the tool. We also thank the anonymous reviewers for their valuable suggestions, which greatly helped to improve the paper. This project has received funding from the European Union Horizon 2020 research and innovation program under grant agreement No 101021727, from ANR grant ANR-20-CE25-0009-TAVA, and from ANR-17-CE25-0014-01 CISC project.

REFERENCES


Manuscript submitted to ACM


[90] [n. d.] HACL*, Lib_Memzero0_memzero function. Retrieved 04/24/2021 from https://github.com/project-everest/hacl-star/blob/v0.3.0/lib/c/Lib_Memzero0.c.


