Binsec/Rel: Symbolic binary analyzer for security with applications to constant-Time and secret-erasure

Lesly-Ann Daniel, Sébastien Bardin, Tamara Rezk

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

LESLY-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:
Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2022. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Trans. Priv. Sec.* 1, 1, Article 1 (January 2022), 44 pages. https://doi.org/10.1145/3563037

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

Authors' addresses: Lesly-Ann Daniel, lesly-ann.daniel@cea.fr, Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France, imec-DistriNet, KU Leuven, Leuven, Belgium; Sébastien Bardin, sebastien.bardin@cea.fr, Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France; Tamara Rezk, tamara.rezk@inria.fr, Inria, Sophia Antipolis, France.

Manuscript submitted to ACM

microarchitectural attacks which requires the control flow and the memory accesses of the program to be independent from the secret input[1]. Constant-time has been proven to protect against cache-based timing attacks [10] and is widely used to secure cryptographic implementations (e.g. BearSSL [11], NaCL [12], HACL* [13], etc). Secret-erasure [14] (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 [15, 16, 17]. 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 [15]. 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 [18, 17, 16], as detailed in CWE-14 [19]. Moreover, these scrubbing operations do not erase secrets that have been copied on the stack by compilers, e.g. from register spilling.

Several CT-analysis tools have been proposed to analyze source code [20, 21], or LLVM code [22, 23], but leave the gap opened for violations introduced in the executable code either by the compiler [15] or by closed-source libraries [24]. Binary-level tools for constant-time using dynamic approaches [25, 26, 27, 28] can find bugs, but otherwise miss vulnerabilities in unexplored portions of the code, making them incomplete. Conversely, static approaches [29, 30, 31] 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 [32, 15] or manual binary-code analysis [18]. While there has been some work on security preserving compilers [17, 15], 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 [9] (i.e., relating two execution traces), and their standard reduction to safety on a transformed program, *self-composition* [33], is inefficient [34];
- Second, it is notoriously difficult to adapt formal methods to binary-level because of the lack of structure information (data and control) and the need to explicitly reason about the memory [35, 36].

A technique that scales well on binary code and that naturally comes into play for bug-finding and bounded-verification is *symbolic execution* (SE) [3, 37]. While it has proven very successful for standard safety properties [38], its direct adaptation to 2-hypersafety properties through (variants of) self-composition suffers from a scalability issue [39, 40, 41]. Some recent approaches scale better, but at the cost of sacrificing either bounded-verification [27, 42] (by doing under-approximations) or bug-finding [23] (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 [43], self-composition [33], product programs [44], multiple facets [45, 46]). In the context of symbolic execution, it has first been coined as *ShadowSE* [47] for back-to-back testing, and later as *relational symbolic execution* (RelSE) [48]. 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.*

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

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 (×1000 speedup) and can achieve bounded-verification when they time out, with a performance close to standard SE (×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 `i686` 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],

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`[2].

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

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

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 $\mathcal{B}v_m$, where m is a constant number, to represent symbolic bitvector expressions of size m. The memory is modeled with a logical array [68] of type $(Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)$ (assuming a 32 bit architecture). A logical array is a function $(Array\ \mathcal{I}\ \mathcal{V})$ that maps each index $i \in \mathcal{I}$ to a value $v \in \mathcal{V}$. Operations over arrays are:

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

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

- $select\ (store\ a\ i\ v)\ i = v$: a store of a value v at index i followed by a *select* at the same index returns the value v;
- $i \neq j \implies select\ (store\ a\ i\ v)\ j = select\ a\ j$: 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 ⟨β|β'⟩ // store high input
2 store ebp-4 ⟨y⟩    // store low input
3 eax := load ebp-4    // assign ⟨λ⟩ to eax
4 ite eax ? l1 : l2    // leak ⟨λ = 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

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: $x = \beta \ \wedge \ y = \lambda \ \wedge \ c = (\lambda \neq 0)$. 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:

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

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 (RelSE)

RelSE improves over self-composition by maximizing *sharing* between the pairs of executions [47, 48]. RelSE models two executions of a program P in the same symbolic execution instance, let us call them p and p'. During RelSE, 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*

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

Fig. 2. RelSE of program in Listing 2 where mem is the memory variable, ebp and eax are registers, $\mu_0, \mu_1, \mu_1', \mu_2, \mu_2'$ are symbolic array variables, and $ebp, \beta, \beta', \lambda, \alpha, \alpha'$ are symbolic bitvector variables

(i.e., the secret input) are represented as *pairs* of expressions. Secret-dependencies are propagated (in a conservative way) through symbolic execution using these relational expressions: if the evaluation of an expression only involves simple operands, its result will be a simple expression, meaning that it does not depend on secret, whereas if it involves a pair of expressions, its result will be a pair of expressions. This representation offers two main advantages. First, this enables sharing redundant parts of p and p', reducing the size of the self-composed formula. Second, variables mapping to simple expressions cannot depend on secret, which makes it possible to spare insecurity queries.

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

Finally, when a control-flow instruction depends on a pair of expressions $\langle \varphi \,|\, \varphi' \rangle$, an insecurity query $\varphi \neq \varphi'$ is sent to the solver. If it is satisfiable, a vulnerability is reported and RelSE continues with the constraint $\varphi = \varphi'$ so the same vulnerability is not reported twice; otherwise the insecurity query is unsatisfiable, meaning that $\varphi = \varphi'$. In both cases, the value of the control-flow instruction is the same in both executions and RelSE only needs to model pairs of executions following the same path.

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

3.3 Challenge of binary-level analysis

Recall that,Binsec/SE represents the memory as a special variable of type $(Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)$. 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.

Version	#Instr	#Query	Time	#Instr/s	Status
SC (e.g. [42])	11k	9051	⌛	3	⌛
RelSE (e.g. [48])	13k	5486	⌛	4	⌛
BINSEC/REL	10M	0	1166	8576	✓

Table 1. Performance of CT-analysis on donna compiled with `gcc-5.4 -O0`, in terms of number of explored unrolled instructions (#Instr), number of queries (#Query), execution time in seconds (Time), instructions explored per second (#Instr/s), and status (Status) set to secure (✓) or timeout (⌛) set to 3600s.

Our solution. To mitigate this issue, we propose dedicated simplifications for binary-level relational symbolic execution that allow a precise tracking of secret-dependencies *in the memory* (details in Section 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 ×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.

$$prog ::= \varepsilon \mid stmt\ prog \qquad stmt ::= < l, inst >$$
$$inst ::= \mathsf{v} := expr \mid \texttt{store}\ expr\ expr \mid \texttt{ite}\ e\ ?\ l_1 : l_2 \mid \texttt{goto}\ expr \mid \texttt{goto}\ l \mid \texttt{halt}$$
$$expr ::= \mathsf{v} \mid \mathsf{bv} \mid \blacktriangleleft expr \mid expr \blacklozenge expr \mid \texttt{load}\ expr$$
$$\blacktriangleleft ::= \neg \mid - \qquad \blacklozenge ::= + \mid \times \mid \leq \mid \ldots$$

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.

Let *Inst* denote the set of instructions and *Loc* the set of program locations. A program $P : Loc \to Inst$ 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,
- $r : Var \to BV_n$ is a register map that maps variables to their bitvector value,
- $m : BV_{32} \to BV_8$ is the memory, mapping 32-bit addresses to bytes and 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 $l \in Loc_\perp \iff P[l] = \texttt{halt}$. For the evaluation of indirect jumps, we define a partial one-to-one correspondence from bitvectors to program locations, $to\_loc : BV_{32} \rightharpoonup Loc$. 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 $(l, r, m)\ e \vdash_t bv$. 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.

Expr

$$\textsc{cst}\ \frac{}{(l,r,m)\ \mathsf{bv} \vdash_\epsilon \mathsf{bv}} \qquad \textsc{var}\ \frac{}{(l,r,m)\ \mathsf{v} \vdash_\epsilon r\ \mathsf{v}} \qquad \textsc{unop}\ \frac{(l,r,m)\ e \vdash_t \mathsf{bv}}{(l,r,m)\ \blacktriangleleft e \vdash_{t\,\cdot\,\lambda_{\blacktriangleleft}(\mathsf{bv})} \blacktriangleleft \mathsf{bv}}$$

$$\textsc{binop}\ \frac{(l,r,m)\ e_1 \vdash_{t_1} \mathsf{bv}_1 \qquad (l,r,m)\ e_2 \vdash_{t_2} \mathsf{bv}_2}{(l,r,m)\ e_1 \blacklozenge e_2 \vdash_{t_1\,\cdot\,t_2\,\cdot\,\lambda_{\blacklozenge}(\mathsf{bv}_1,\mathsf{bv}_2)} \mathsf{bv}_1 \blacklozenge \mathsf{bv}_2} \qquad \textsc{load}\ \frac{(l,r,m)\ e \vdash_t \mathsf{bv}}{(l,r,m)\ \texttt{load}\ e \vdash_{t\,\cdot\,\lambda_{@}(\mathsf{bv})} m\ \mathsf{bv}}$$

Instr

$$\textsc{halt}\ \frac{P[l] = \texttt{halt}}{(l,r,m) \xrightarrow[\lambda_\bot(l)]{} (l,r,m)} \qquad \textsc{s\_jump}\ \frac{P[l] = \texttt{goto}\ l'}{(l,r,m) \xrightarrow[\lambda_{pc}(l')]{} (l',r,m)}$$

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

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

$$\textsc{ite-false}\ \frac{P[l] = \texttt{ite}\ e\ \texttt{?}\ l_1\ \texttt{:}\ l_2 \qquad (l,r,m)\ e \vdash_t \mathsf{bv} \qquad \mathsf{bv} = 0}{(l,r,m) \xrightarrow[t\,\cdot\,\lambda_{pc}(l_2)]{} (l_2,r,m)} \qquad \textsc{assign}\ \frac{P[l] = \mathsf{v} := e \qquad (l,r,m) e \vdash_t \mathsf{bv}}{(l,r,m) \xrightarrow[t]{} (l+1, r[\mathsf{v} \mapsto \mathsf{bv}], m)}$$

$$\textsc{store}\ \frac{P[l] = \texttt{store}\ e\ e' \qquad (l,r,m)\ e \vdash_t \mathsf{bv} \qquad (l,r,m)\ e' \vdash_{t'} \mathsf{bv}'}{(l,r,m) \xrightarrow[t'\,\cdot\,t\,\cdot\,\lambda_{@}(\mathsf{bv})\,\cdot\,\lambda_{\mu}(\mathsf{bv},\mathsf{bv}')]{} (l+1, r, m[\mathsf{bv} \mapsto \mathsf{bv}'])}$$

Fig. 4. Concrete evaluation of DBA instructions and expressions.

The concrete semantics is given in Fig. 4 and is parameterized with leakage functions $\lambda_{\blacktriangleleft} : BV \to \mathcal{L}$, $\lambda_{\blacklozenge} : BV \times BV \to \mathcal{L}$, $\lambda_{@} : BV_{32} \to \mathcal{L}$, $\lambda_{pc} : Loc \to \mathcal{L}$, $\lambda_{\perp} : Loc \to \mathcal{L}$, $\lambda_{\mu} : BV_{32} \times BV_8 \to \mathcal{L}$. 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: $\mathcal{L} \triangleq List(Loc)$. 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: $\mathcal{L} \triangleq List(BV_{32})$. The addresses of memory load and stores are leaked: $\lambda_{@}(e) = [e]$. 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: $\mathcal{L} \triangleq List(BV)$. Functions $\lambda_{\blacktriangleleft}$ and $\lambda_{\blacklozenge}$ 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 $\lambda_{\div}(bv_1, bv_2) = [size(bv_1), size(bv_2)]$.

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 $\mathcal{L} \triangleq List(Loc \cup BV_{32})$. The control flow is leaked $\lambda_{pc}(l) = [l]$, as well as the memory accesses $\lambda_{@}(e) = [e]$. 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: $\mathcal{L} \triangleq List(Loc \cup (BV_{32} \times BV_8))$. 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 $H_v \subseteq Var$ be the set of high (secret) variables and $L_v = Var \setminus H_v$ be the set of low (public) variables. Analogously, we define $H_{@} \subseteq BV_{32}$ (resp. $L_{@} = BV_{32} \setminus H_{@}$) as the addresses containing high (resp. low) input in the initial memory. The *low-equivalence relation* over concrete configurations c and c', denoted $c \simeq_L c'$, is defined as the equality of low variables and low parts of the memory. Formally, two configurations $c \triangleq (l, r, m)$ and $c' \triangleq (l', r', m')$ are low-equivalent if and only if for all variable $v \in L_v$, $r\ v = r'\ v$ and for all address $a \in L_{@}$, $m\ a = m'\ a$.

[3]See https://bearssl.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 $k \in \mathbb{N}$,*

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

The property is parameterized by a function, $filter : \mathcal{L} \to \mathcal{L}$, 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 ($l \in Loc_\perp$), restricting to values that have not been overwritten by a more recent store. Formally, $filter(t) = filter'(t, m_\varepsilon)$ where m_ε is the empty partial function from BV_{32} to BV_8 and $filter'(t, m_{acc})$ is defined as:*

$$\textsc{filter-empty}\ filter'(\varepsilon, m_{acc}) = \varepsilon \qquad \textsc{filter-store}\ filter'((\mathtt{a}, \mathtt{v}) \cdot t, m_{acc}) = filter'(t, m_{acc}[\mathtt{a} \mapsto \mathtt{v}])$$

$$\textsc{filter-cf}\ \frac{l \notin Loc_\perp}{filter'(l \cdot t, m_{acc}) = l \cdot filter'(t, m_{acc})} \qquad \textsc{filter-halt}\ \frac{\mathtt{a_i} \in dom(m_{acc}) \qquad l \in Loc_\perp}{filter'(l \cdot t, m_{acc}) = m_{acc}(\mathtt{a_0}) \cdot \ldots \cdot m_{acc}(\mathtt{a_n})}$$

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 $m_{acc}(\mathtt{a})$ is already defined, its value will be replaced by $\mathtt{v}$ after $m_{acc}[\mathtt{a} \mapsto \mathtt{v}]$. 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, $filter((\mathtt{a}, \mathtt{x}) \cdot (\mathtt{b}, \mathtt{y}) \cdot (\mathtt{a}, \mathtt{z}) \cdot l_\perp)$ where $l_\perp \in Loc_\perp$ will return the leakage $\mathtt{y} \cdot \mathtt{z}$.*

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 β, β', λ, φ, range over the set of formulas Φ in the QF_ABV logic. A *relational* formula $\widehat{\varphi}$ is either a QF_ABV formula $\langle\varphi\rangle$ or a pair $\langle\varphi_l \,|\, \varphi_r\rangle$ of two QF_ABV formulas. We denote $\widehat{\varphi}_{|l}$ (resp. $\widehat{\varphi}_{|r}$), the projection on the left (resp. right) value of $\widehat{\varphi}$. If $\widehat{\varphi} = \langle\varphi\rangle$, then $\widehat{\varphi}_{|l}$ and $\widehat{\varphi}_{|r}$ are both defined as φ. Let $\mathbf{\Phi}$ be the set of relational formulas and $\boldsymbol{\mathcal{B}v_n}$ 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 $l \in Loc$ at any point of the execution. A *symbolic configuration* is of the form $(l, \rho, \widehat{\mu}, \pi)$ where:

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

- $\hat{\mu} : (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8) \times (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)$ is the symbolic memory—a pair of arrays of values in $\mathcal{B}v_8$ indexed by addresses in $\mathcal{B}v_{32}$,
- $\pi \in \Phi$ is the path predicate—a conjunction of conditional statements and assignments encountered along a path.

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

The symbolic evaluation is parameterized by *symbolic leakage predicates* $\tilde{\lambda}_{\blacklozenge}, \tilde{\lambda}_{\blacklozenge}, \tilde{\lambda}_{@}, \tilde{\lambda}_{dj}, \tilde{\lambda}_{ite}$ and $\tilde{\lambda}_{\bot}$ 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 $true$ 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 $true$, meaning that no secret can leak. If a symbolic leakage predicate evaluates to $false$ 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 *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 *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 $true$ (the $false$ case is analogous). If the condition guarding the $true$-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 $true$ 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 $\tilde{\lambda}_{\blacklozenge}, \tilde{\lambda}_{\blacklozenge}, \tilde{\lambda}_{@}, \tilde{\lambda}_{dj}, \tilde{\lambda}_{ite}$ and $\tilde{\lambda}_{\bot}$ according to the leakage model (cf. Section 5.1.2).

Expr

$$\textsc{cst}\ \frac{}{(\rho,\widehat{\mu},\pi)\ \mathsf{bv} \vdash \langle bv\rangle} \qquad \textsc{var}\ \frac{}{(\rho,\widehat{\mu},\pi)\ \mathsf{v} \vdash \rho\ \mathsf{v}}$$

$$\textsc{unop}\ \frac{(\rho,\widehat{\mu},\pi)\ e \vdash \widehat{\phi} \qquad \widehat{\varphi} \triangleq \triangleleft\, \widehat{\phi} \qquad \tilde{\lambda}_{\blacktriangleleft}(\pi,\widehat{\phi})}{(\rho,\widehat{\mu},\pi) \blacktriangleleft e \vdash \widehat{\varphi}}$$

$$\textsc{binop}\ \frac{(\rho,\widehat{\mu},\pi)\ e_1 \vdash \widehat{\phi} \qquad (\rho,\widehat{\mu},\pi)\ e_2 \vdash \widehat{\psi} \qquad \widehat{\varphi} \triangleq \widehat{\phi} \diamond \widehat{\psi} \qquad \tilde{\lambda}_{\blacklozenge}(\pi,\widehat{\phi},\widehat{\psi})}{(\rho,\widehat{\mu},\pi)\ e_1 \blacklozenge e_2 \vdash \widehat{\varphi}}$$

$$\textsc{load}\ \frac{(\rho,\widehat{\mu},\pi)\ e_{idx} \vdash \hat{\imath} \qquad \boxed{\widehat{\varphi} \triangleq \langle select(\widehat{\mu}_{|l}, \hat{\imath}_{|l}) \,|\, select(\widehat{\mu}_{|r}, \hat{\imath}_{|r})\rangle} \qquad \tilde{\lambda}_{@}(\pi,\hat{\imath})}{(\rho,\widehat{\mu},\pi)\ \texttt{load}\ e_{idx} \vdash \widehat{\varphi}}$$

Instr

$$\textsc{halt}\ \frac{P[l] = \texttt{halt} \qquad \tilde{\lambda}_{\perp}(\pi,\widehat{\mu})}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l,\rho,\widehat{\mu},\pi)} \qquad \textsc{s\_jump}\ \frac{P[l] = \texttt{goto}\ l'}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l',\rho,\widehat{\mu},\pi)}$$

$$\textsc{d\_jump}\ \frac{\begin{array}{c}P[l] = \texttt{goto}\ e \qquad (\rho,\widehat{\mu},\pi)\ e \vdash \widehat{\varphi} \\ M \vDash_{\text{SMT}} \pi \wedge \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r} \qquad l' \triangleq to\_loc(M(\widehat{\varphi}_{|l})) \qquad \pi' \triangleq \pi \wedge (\widehat{\varphi}_{|l} = \widehat{\varphi}_{|r} = M(\widehat{\varphi}_{|l})) \qquad \tilde{\lambda}_{dj}(\pi,\widehat{\varphi})\end{array}}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l',\rho,\widehat{\mu},\pi')}$$

ITE-TRUE

$$\frac{\begin{array}{c}P[l] = \texttt{ite}\ e\ ?\ l_{true} : l_{false} \\ l' \triangleq l_{true} \qquad (\rho,\widehat{\mu},\pi)\ e \vdash \widehat{\varphi} \qquad \pi' \triangleq \pi \wedge (\widehat{\varphi}_{|l} \neq 0) \wedge (\widehat{\varphi}_{|r} \neq 0) \qquad \vDash_{\text{SMT}} \pi' \qquad \tilde{\lambda}_{ite}(\pi,\widehat{\varphi})\end{array}}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l',\rho,\widehat{\mu},\pi')}$$

ITE-FALSE

$$\frac{\begin{array}{c}P[l] = \texttt{ite}\ e\ ?\ l_{true} : l_{false} \\ l' \triangleq l_{false} \qquad (\rho,\widehat{\mu},\pi)\ e \vdash \widehat{\varphi} \qquad \pi' \triangleq \pi \wedge (\widehat{\varphi}_{|l} = 0) \wedge (\widehat{\varphi}_{|r} = 0) \qquad \vDash_{\text{SMT}} \pi' \qquad \tilde{\lambda}_{ite}(\pi,\widehat{\varphi})\end{array}}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l',\rho,\widehat{\mu},\pi')}$$

$$\textsc{assign}\ \frac{\begin{array}{c}P[l] = \mathsf{v} := e \qquad (\rho,\widehat{\mu},\pi)\ e \vdash \widehat{\varphi} \\ \boxed{\neg canonical(\widehat{\varphi})} \qquad \widehat{\varphi}' \triangleq fresh \qquad \rho' \triangleq \rho[v \mapsto \widehat{\varphi}'] \qquad \pi' \triangleq \pi \wedge (\widehat{\varphi}'_{|l} = \widehat{\varphi}_{|l}) \wedge (\widehat{\varphi}'_{|r} = \widehat{\varphi}_{|r})\end{array}}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l+1,\rho',\widehat{\mu},\pi')}$$

$$\textsc{canonical-assign}\ \frac{P[l] = \mathsf{v} := e \qquad (\rho,\widehat{\mu},\pi)\ e \vdash \widehat{\varphi} \qquad canonical(\widehat{\varphi}) \qquad \rho' \triangleq \rho[v \mapsto \widehat{\varphi}]}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l+1,\rho',\widehat{\mu},\pi)}$$

STORE

$$\frac{\begin{array}{c}P[l] = \texttt{store}\ e_{idx}\ e_{val} \\ (\rho,\widehat{\mu},\pi)\ e_{idx} \vdash \hat{\imath} \qquad (\rho,\widehat{\mu},\pi)\ e_{val} \vdash \hat{v} \qquad \widehat{\mu}' \triangleq \langle store(\widehat{\mu}_{|l}, \hat{\imath}_{|l}, \hat{v}_{|l}) \,|\, store(\widehat{\mu}_{|r}, \hat{\imath}_{|r}, \hat{v}_{|r})\rangle \\ \pi' \triangleq \pi \wedge \widehat{\mu}'_{|l} = store(\widehat{\mu}_{|l}, \hat{\imath}_{|l}, \hat{v}_{|l}) \wedge \widehat{\mu}'_{|r} = store(\widehat{\mu}_{|r}, \hat{\imath}_{|r}, \hat{v}_{|r}) \qquad \tilde{\lambda}_{@}(\pi,\hat{\imath})\end{array}}{(l,\rho,\widehat{\mu},\pi) \rightsquigarrow (l+1,\rho,\widehat{\mu}',\pi')}$$

Fig. 5. Symbolic evaluation of DBA instructions and expressions where *fresh* returns a fresh symbolic variable, $canonical(\widehat{\varphi})$ is $true$ if $\widehat{\varphi}$ is in canonical form; and $\triangleleft$ (resp. $\diamond$) is the logical counterpart of the concrete operator $\blacktriangleleft$ (resp. $\blacklozenge$), lifted to relational expressions.

5.1.1 Predicate secLeak. We define a predicate $secLeak : \Phi \times \Phi \rightarrow Bool$ which ensures that a relational formula does not differ in its right and left components, meaning that it can be leaked securely:

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

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: $\tilde{\lambda}_{dj}(\pi, \widehat{\varphi}) = secLeak(\widehat{\varphi}, \pi)$ and $\tilde{\lambda}_{ite}(\pi, \widehat{\varphi}) = secLeak(\boldsymbol{eq}_0\ \widehat{\varphi}, \pi)$ where $eq_0\ x$ returns $true$ if $x = 0$ and $false$ otherwise, and $\boldsymbol{eq}_0$ is the lifting of 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: $\tilde{\lambda}_{@}(\pi, \widehat{\varphi}) = secLeak(\widehat{\varphi}, \pi)$. 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, $\tilde{\lambda}_{<<}(\pi, \widehat{\varphi}, \widehat{\phi}) = secLeak(\widehat{\varphi}, \pi)$. 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 $\tilde{\lambda}_{\div}(\pi, \widehat{\varphi}, \widehat{\psi}) = secLeak(\boldsymbol{size}\ \widehat{\varphi}, \pi)$, where $size : \mathcal{B}v \rightarrow \mathcal{B}v$ is a function that returns the size of a symbolic bitvector and $\boldsymbol{size}$ 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 $\tilde{\lambda}_{dj}$ and $\tilde{\lambda}_{ite}$ are defined like in the program counter policy, while $\tilde{\lambda}_{@}$ 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:

$$\tilde{\lambda}_{\perp}(\pi, \widehat{\mu}) = \bigwedge_{\iota \in addr(\widehat{\mu})} secLeak(\langle select(\widehat{\mu}_{|l}, \iota) \,|\, select(\widehat{\mu}_{|r}, \iota) \rangle, \pi)$$

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

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 $secLeak(\widehat{\varphi}, \pi)$ evaluates to *false*. In this case, the insecurity query is satisfiable and the solver returns a model M such that $M \vDash_{\text{SMT}} \pi \wedge (\widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$. The model M assigns concrete values to variables that satisfy the insecurity query. Therefore it can be returned as a concrete counterexample 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 μ_0. Therefore, a memory *select* can be resolved by going back up the history and comparing the index to load, with indexes previously stored. Our FlyRow optimization consists in replacing selection in the memory (Figure 5, LOAD rule, boxed hypothesis) by a new function $lookup : ((Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8) \times (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)) \times \boldsymbol{\mathcal{B}v}_{32} \rightarrow \boldsymbol{\mathcal{B}v}_8$ 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]:

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

$$lookup(\widehat{\mu}_n, \iota) = \begin{cases} \langle \varphi_l \rangle & \text{if } eq^{\#}(\iota, \kappa) \wedge eq^{\#}(\varphi_l, \varphi_r) \\ \langle \varphi_l \,|\, \varphi_r \rangle & \text{if } eq^{\#}(\iota, \kappa) \wedge \neg eq^{\#}(\varphi_l, \varphi_r) \\ lookup(\widehat{\mu}_{n-1}, \iota) & \text{if } \neg eq^{\#}(\iota, \kappa) \\ \langle select(\widehat{\mu}_{n\,|l}, \iota) \,|\, select(\widehat{\mu}_{n\,|r}, \iota)\rangle & \text{if } eq^{\#}(\iota, \kappa) = \bot \end{cases}$$

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

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

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

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

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

Term rewriting. To improve the conclusiveness of syntactic equality checks for the read-over-write, the terms are assumed to be in *normalized* form $\beta + o$ where β is a base (i.e., an expression on symbolic variables) and o is a constant offset. The comparison of two terms $\beta + o$ and $\beta' + o'$ in normalized form can be efficiently computed as follows: if the bases β and β' 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).

$$normalize\,(c + t) = t + c \qquad normalize\,(-(t + c)) = (-t) - c$$
$$normalize\,((t + c) + c') = t + (c + c') \qquad normalize\,((t + c) + t') = (t + t') + c$$
$$normalize\,((t + c) + (t' + c')) = (t + t') + (c + c') \qquad normalize\,((t + c) - (t' + c')) = (t - t') + (c - c')$$

Fig. 6. Rewriting rules for normalization (non exhaustive). All expressions belong to the set $\mathcal{B}v$ 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). $normalize\,((eax + 4) + (ebx + 4)) = (eax + ebx) + 8$

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

We define in Fig. 7 a function $untaint(\rho, \widehat{\mu}, \widehat{\varphi})$ which takes a register map ρ, a memory $\widehat{\mu}$, and a duplicated expression $\widehat{\varphi}$. It deduces variable equalities from $\widehat{\varphi}$, propagate them in ρ and $\widehat{\mu}$, and returns a pair of updated register map and memory $(\rho', \widehat{\mu}')$. Intuitively, if the equality of variables x_l and x_r can be deduced from $secLeak(\widehat{\varphi}, \pi)$, the *untaint* function replaces occurrences of x_r by x_l in the memory and the register map. As a result, a duplicated expression $\langle x_l \,|\, x_r \rangle$ would be replaced by the simple expression $\langle x_l \rangle$ in the rest of the execution[4].

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

Fig. 7. Untainting rules where x_l, x_r are bitvector variables of the same size, t_l, t_r, k, l are bitvector terms such that t_l, t_r, k have the same size, :: indicates the concatenation of bitvectors, and $f[x_r \backslash x_l]$ indicates that the variable x_r is substituted with x_l 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 $\langle \varphi_{|l} \,|\, \varphi_{|r} \rangle$ and $\langle \phi_{|l} \,|\, \phi_{|r} \rangle$, we would normally generate two insecurity queries $(\pi \wedge \varphi_{|l} \neq \varphi_{|r})$ and $(\pi \wedge \phi_{|l} \neq \phi_{|r})$—one for each memory access. Fault-packing regroups these checks into a single query $\big(\pi \wedge ((\varphi_{|l} \neq \varphi_{|r}) \vee (\phi_{|l} \neq \phi_{|r}))\big)$ sent to the solver at the end of the basic block.

This optimization reduces the number of insecurity queries sent to the solver and thus helps improving performance. However it degrades the precision of the counterexample: while checking each instruction individually precisely points to vulnerable instructions, fault-packing reduces accuracy to vulnerable basic blocks only. Note that even though disjunctive constraints are usually harder to solve than pure conjunctive constraints, those introduced by *FP* are very 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.

[4]We 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. $\rightsquigarrow^k$), 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 φ, $M \vDash \pi \iff M \vDash_{SMT} \pi$.*

Hypothesis 3. *We assume that the program P is defined on all locations computed during the symbolic execution—notably by the function to_loc in rule* D_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 $\widehat{\varphi}$ is leaked such that $secLeak(\widehat{\varphi}, \pi)$ evaluates to $false$ and the solver returns a model M such that $M \vDash \pi \wedge (\widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$ (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 ($\approx_p^M$). *We define a concretization relation $\approx_p^M$ between concrete and symbolic configurations, where M is a model and $p \in \{l, r\}$ is a projection on the left or right side of a symbolic configuration. Intuitively, the relation $c \approx_p^M s$ is the concretization of the p-side of the symbolic state s with the model M. Let $c \triangleq (l_1, r, m)$ and $s \triangleq (l_2, \rho, \widehat{\mu}, \pi)$. Formally $c \approx_p^M s$ holds iff $M \vDash \pi$, $l_1 = l_2$ and for all expression e, either the symbolic evaluation of e gets stuck or we have*

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

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 $c_0 \approx_l^M s_0 \wedge c_0' \approx_r^M s_0$, then $c_0 \simeq_L c_0'$.*

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 $c_k \approx_l^M s_k$ and $c_k' \approx_r^M s_k$, 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 $c_k \xrightarrow[t]{} c_{k+1}$ and $c_k' \xrightarrow[t']{} c_{k+1}'$. 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 $c_0 \approx_l^M s_0 \wedge c'_0 \approx_r^M s_0$, if $c_0 \xrightarrow[t]{}^k c_k$ and $c'_0 \xrightarrow[t']{}^k c'_k$ follow the same path, then there exists a symbolic configuration s_k and a model M' such that:*

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

Proof 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} to a state 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 $s_0 \rightsquigarrow^k s_k$ and for every concrete configurations c_0, c_k and model M, such that $c_0 \approx_p^M s_0$ and $c_k \approx_p^M s_k$, there exists a concrete execution $c_0 \rightarrow^k c_k$.*

Proof overview: The proof goes by induction on the number of steps k. For each symbolic step $s_{k-1} \rightsquigarrow s_k$ and model M_k such that $c_{k-1} \approx_p^{M_k} s_{k-1}$ and $c_k \approx_p^{M_k} s_k$, 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} to a state c'_k (from Hypothesis 4 and Proposition 2), and, because there is no over-approximation in symbolic execution, c'_k satisfies $c'_k \approx_p^{M_k} s_k$. □

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 $s_0 \rightsquigarrow^k s_k$ such that s_k is stuck, then there exists a model M and concrete configurations $c_0 \approx_l^M s_0, c'_0 \approx_r^M s_0, c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$ such that,*

$$c_0 \simeq_L c'_0 \wedge c_0 \xrightarrow[t]{}^k c_k \xrightarrow[t_k]{} c_{k+1} \wedge c'_0 \xrightarrow[t']{}^k c'_k \xrightarrow[t'_k]{} c_{k+1} \wedge t_k \neq t'_k$$

Proof: Let us consider symbolic configurations s_0 and s_k such that $s_0 \rightsquigarrow^k s_k$ and s_k is stuck. From Lemma 1, there is a model M and concrete configurations c_k and c'_k such that $c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$, and $c_k \xrightarrow[t_k]{} c_{k+1}$ and $c'_k \xrightarrow[t'_k]{} c'_{k+1}$ with $t_k \neq t'_k$. Additionally, let c_0, c'_0 be concrete configurations such that $c_0 \approx_l^M s_0$ and $c'_0 \approx_r^M s_0$. From Proposition 3, we have $c_0 \simeq_L c'_0$, and from Theorem 1, there are concrete executions $c_0 \xrightarrow[t]{}^k c_k$ and $c'_0 \xrightarrow[t']{}^k c'_k$. Therefore, we have $c_0 \xrightarrow[t]{}^k c_k \xrightarrow[t_k]{} c_{k+1}$ and $c'_0 \xrightarrow[t']{}^k c'_k \xrightarrow[t'_k]{} c'_{k+1}$ with $c_0 \simeq_L c'_0$ 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 $c_0 \approx_l^M s_0 \wedge c'_0 \approx_r^M s_0$, if $c_0 \xrightarrow[t]{}^k c_k$ and $c'_0 \xrightarrow[t]{}^k c'_k$ then there exists a symbolic configuration s_k and a model M' such that:*

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

Proof: 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, $s_0 \rightsquigarrow^k s_k$ then for all initial configurations c_0 and c'_0 and model M such that $c_0 \approx_l^M s_0$, and $c'_0 \approx_r^M s_0$,*

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

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

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, $\#I_{unr}$ 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 libsodium [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 RelSE (see Section 6.3). *Moreover, this is the first automatic constant-time analysis of these cryptographic libraries at the binary-level.*

Description and number of binaries[†]		≈ #I	#I_{unr}	P	Time	Status	Type	Key	Msg
Utility	ct-select (×29)	1015	1507	29	0.2	29 × ✓	Utility functions	-	9
	ct-sort (×12)	2400	1782	12	0.2	12 × ✓		-	12
	Hacl* (×110)	3850	90953	110	7.6	110 × ✓		-	2-200
	OpenSSL (×130)	4550	5113	130	0.9	130 × ✓		-	4-12
Tea	decrypt `-O0`	290	953	1	0.1	✓	Block cipher	16	8
	decrypt `-O3`	250	804	1	0.1	✓			
Donna	`-O0`	7083	10.2M	1	1008.5	✓	Elliptic curve	32	-
	`-O3`	4643	2.7M	1	347.1	✓			
Libsodium	salsa20	1627	38.0k	1	3.5	✓	Stream cipher	32	256
	chacha20	2717	12.3k	1	1.5	✓	Stream cipher	32	256
	sha256	4879	48.4k	1	4.5	✓	Secure hash	-	256
	sha512	16312	92.0k	1	8.1	✓	Secure hash	-	256
Hacl*	chacha20	1221	6.7k	1	4.3	✓	Stream cipher	32	256
	sha256	1279	21.0k	1	2.7	✓	Secure hash	-	256
	sha512	2013	47.5k	1	5.2	✓	Secure hash	-	256
	curve25519	8522	9.4M	1	927.8	✓	Elliptic curve	32	-
BearSSL	aes-ct-cbcenc[‡]	357	3.5k	1	0.5	✓	Block cipher	240	32
	des-ct-cbcenc[‡]	682	19.9k	1	12.1	✓	Block cipher	384	16
OpenSSL	tls-remove-padding-patch	424	35.7k	520	438.0	✓	Remove padding	-	63
Total	296 binaries	64114	22.8M	815	2772.7	296 × ✓	-	-	-

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-padding`[5] 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.

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

Description and nb. of binaries†		≈ #I	#I_{unr}	P	Time	CT	Status	🐞	Type	Key	Msg
Utility	ct-select (×21)	735	767	42	0.4	Y	21 × ✗ (1 new)	21	Utility func.	-	-
	ct-sort (×18)	3600	7513	138	13.6	Y	18 × ✗ (2 new)	44		-	-
BearSSL	aes-big-cbcenc‡	375	876	1	1651.8	N	✗	32	Block cipher	240	32
	des-tab-cbcenc‡	365	5187	1	4.4	N	✗	8	Block cipher	384	16
OpenSSL	tls-rem-pad-lucky13	950	7866	375	700.3	N	✗	6	Remove pad.	-	63
Total	42 binaries	6025	22209	557	2370.5	-	42 × ✗	111	-	-	-

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 setups[6]. 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

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

compiler	clang-3.0	clang-3.9	clang-7.1	clang-7.1	clang-9/11	clang-9/11	gcc-all	gcc-all	gcc-10.3
arch	i386/i686	i386/i686	i386	i686	i386	i686	i386	i686	arm
opt-level	0 1 2 3 3⁻	0 1 2 3 3⁻	0 1 2 3 3⁻	0 1 2 3 3⁻	0 1 2 3 3⁻	0 1 2 3 3⁻	0 0⁺ 1 2 3 3⁻	0 0⁺ 1 2 3 3⁻	0 0⁺ 1 2 3 3⁻
ct_select_v1	✓✓ c c -	✓✓ c c -	✓✓ c c c	✓✓ M M M	✓✓ c c c	✓✓ M M M	✓✓✓✓✓✓	✓✓✓✓✓✓	✓✓✓✓✓✓
ct_select_v2	✓ c c c -	✓ c c c -	✓ c c c c	✓ M M M M	✓ c c c c	✓ M M M M	✓✓✓✓✓✓	✓✓✓✓✓✓	✓✓✓✓✓✓
ct_select_v3	✓✓✓✓ -	✓ c c c -	✓ c c c c	✓ M M M M	✓ c c c c	✓ M M M M	✓✓✓✓✓✓	✓✓✓✓✓✓	✓✓✓✓✓✓
ct_select_v4	✓ c c c -	✓ c c c -	✓ c c c c	✓ M M M M	✓ c c c c	✓ M M M M	✓✓✓✓✓✓	✓✓✓✓✓✓	✓✓✓✓✓✓
naive_select	c c c c -	c c c c -	c c c c c	c M M M M	c c c c c	c M M M M	c c c ✓✓✓ c	c c ✓✓✓ c	c c ✓✓✓ c
sort	✓ c c c -	✓ c c c -	✓✓✓✓✓	✓✓✓✓✓	✓ c c c c	✓ M M M M	c c ✓✓✓✓	c c ✓✓✓✓	c c ✓✓✓✓
sort_multiplex	✓ c c c -	✓ c c c -	✓✓✓✓✓	✓✓✓✓✓	✓ c c c c	✓ M M M M	c c ✓✓✓✓	c c ✓✓✓✓	c c ✓✓✓✓
sort_naive	c c c c -	c c c c -	c c c c c	c M M M M	c c c c c	c M M M M	c c c c c c	c c c c c c	c c ✓✓✓ c
HACL*-utility ×11	✓✓✓✓ -	✓✓✓✓ -	✓✓✓✓✓	✓✓✓✓✓	✓✓✓✓✓	✓✓✓✓✓	✓✓✓✓✓✓	✓✓✓✓✓✓	✓✓✓✓✓✓
OpenSSL-utility ×13	✓✓✓✓ -	✓✓✓✓ -	✓✓✓✓✓	✓✓✓✓✓	✓✓✓✓✓	✓✓✓✓✓	✓✓✓✓✓✓	✓✓✓✓✓✓	✓✓✓✓✓✓
tea-enc/dec ×2	✓✓✓✓ -	✓✓✓✓ -	✓✓✓✓✓	✓✓✓✓✓	✓✓✓✓✓	✓✓✓✓✓	✓✓✓✓✓✓	✓✓✓✓✓✓	✓✓✓✓✓✓

Table 4. Preservation of constant-time for several programs compiled with gcc or clang for i386, i686 or arm architectures and optimization levels O0, O0⁺ O1, O2, O3 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.

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 ct-verif [22] 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[7];
- 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.

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.

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

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 preformance 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 $\#I_{unr}$, number of instruction explored per seconds ($\#I_{unr}/s$), 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.

	$\#I_{unr}$	$\#I_{unr}/s$	#Q	$\#Q_e$	$\#Q_i$	Time	⧗	✓	✗
SC	248k	3.9	158k	14k	143k	64296	16	280	42
RelSE	349k	6.2	90k	17k	73k	56428	13	283	42
Binsec/Rel	22.8M	6238	3700	2408	1292	3657	0	296	42

Table 5. Binsec/Rel vs. standard approaches.

While *RelSE* performs slightly better than *SC* (1.6× speedup in terms of $\#I_{unr}/s$) 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 $\#I_{unr}/s$;
- 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 $\#I_{unr}/s$;
- 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

	Version	#I$_{unr}$	#I$_{unr}$/s	#Q	#Q$_e$	#Q$_i$	Time	⧗	✓	✗
Standard RelSE (without *FlyRow*)	*RelSE*	349k	6.2	90148	17428	72720	56429	13	283	42
	+ *Unt*	414k	9.9	48648	20601	28047	41852	7	289	42
	+ *FP*	437k	12.7	35100	21834	13266	34471	7	289	42
Binsec/Rel (with *FlyRow*)	*RelSE + FlyRow*	22.8M	4450	3738	2408	1330	5127	0	296	42
	+ *Unt*	22.8M	4429	3738	2408	1330	5151	0	296	42
	+ *FP*	22.8M	6238	3700	2408	1292	3658	0	296	42

Table 6. Performances of Binsec/Rel simplifications.

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*.

Version	#I$_{unr}$	#I$_{unr}$/s	#Q	Time	⧗
SE	522k	19.5	24444	26814	6
SE+PostRow [74]	628k	29.2	29389	21475	4
SE+FlyRow	22.8M	12531.1	534	1817	0

Version	#I$_{unr}$	#I$_{unr}$/s	#Q	Time	⧗
RelSE	349k	6.2	90148	56428	13
RelSE+PostRow	317k	5.3	65834	60295	15
Binsec/Rel	22.8M	6237.7	3700	3657	0

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

- Binsec/Rel, compared to our best setting for symbolic execution (*SE+FlyRow*), only has an overhead of 2× in terms of #I$_{unr}$/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.

	clang-all-versions				gcc-5.4/6.2/7.2				gcc-8.3/10.2			
	`-O0`	`-O1`	`-O2`	`-O3`	`-O0`	`-O1`	`-O2`	`-O3`	`-O0`	`-O1`	`-O2`	`-O3`
`loop`	✓	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗	✗
`memset`	✓	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗	✗
`bzero`	✓	✓	✗	✗	✓	(✗)	✗	✗	✓	(✓)	✗	✗

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.

```c
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.

clang-all-versions				gcc-all-versions			
-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3
✓	✓	✓	✓	✓	✓	✗	✗

Table 9. Preservation of secret-erasure with volatile function pointer as implemented in Listing 3.

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`.

```c
1 volatile char *vbuf = (volatile char *) buf; // Pointer-to-volatile
2 char * volatile vbuf = (char *volatile) buf; // Volatile ptr (pointer is volatile itself)
3 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

[10] Although we did not find this implementation in real-world cryptographic code, we were curious about how the compiler would handle this case.

	clang-all-versions				gcc-all-versions			
	-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3
ptr_to_volatile_loop	✓	✓	✓	✓	✓	✓	✓	✓
volatile_ptr_loop	✓	✓	✓	✓	✓	✓	✗	✗
vol_ptr_to_vol_loop	✓	✓	✓	✓	✓	✓	✓	✓
ptr_to_volatile_memset	✓	✓	✗	✗	✓	✓	✗	✗
volatile_ptr_memset	✓	✓	✓	✓	✓	✓	✗	✗
vol_ptr_to_vol_memset	✓	✓	✓	✓	✓	✓	✗	✗

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

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.

```
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 5. 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][11].

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 libsodium `memzero` [89] when weak symbols are available.

```
__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 a `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-11.0.1`[12], 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-erasure[13].

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.

[12] `clang-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.
[13] However, we strongly suspect that this conclusion does not generalize to all programs, for instance to programs that violate secret-erasure because of register spilling.

	gcc-5.4.0 6.2.0 7.2.0				gcc-8.3.0 10.2.0			
	-O3	dse	tdse	both	-O3	dse	tdse	both
loop	✗	✗	✗	✗	✗	✗	✗	✗
memset	✗	✗	✗	(✓)	✗	✗	✗	(✓)
bzero	✗	(✓)	✗	✓	✗	✗	✗	(✓)
explicit_bzero	✓	✓	✓	✓	✓	✓	✓	✓
memset_s	✓	✓	✓	✓	✓	✓	✓	✓
volatile_func	✗	✗	✗	✗	✗	✗	✗	✗
ptr_to_volatile_loop	✓	✓	✓	✓	✓	✓	✓	✓
ptr_to_volatile_memset	✗	✗	✗	(✓)	✗	✗	✗	(✓)
volatile_ptr_loop	✗	✗	✗	✗	✗	✗	✗	✗
volatile_ptr_memset	✗	✗	(✓)	✓	✗	✗	(✓)	✓
vol_ptr_to_vol_loop	✓	✓	✓	✓	✓	✓	✓	✓
vol_ptr_to_vol_memset	✗	✗	(✓)	✓	✗	✗	(✓)	✓
memory_barrier_all (×4)	✓	✓	✓	✓	✓	✓	✓	✓
weak_symbols	✓	✓	✓	✓	✓	✓	✓	✓

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.

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[14]. A possible solution to enable sound analysis for program with unbounded loops would be to use relational loop invariants [39]—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[15]. 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 [95], however the technique has recently been adapted to a speculative semantics [51].

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] 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] 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].

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

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_s for static instructions, and I_u 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 *relational symbolic execution* has been coined more recently [48] but this work is limited to a simple toy imperative language and do not address scalability.

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

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. [27] and Subramanyan *et al.* [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 *et al.* [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.

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.

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 [102], dynamic binary instrumentation [25, 28], dynamic symbolic execution (DSE) [26], or fuzzing [103].

Static approaches based on sound static analysis [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 [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 [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.

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

Chong and Myers [14] introduce the first framework to specify erasure policies which has been later refined to express richer policies using a knowledge-based approach [117], and cryptographic data deletion [118]. These works focus on expressing complex secret-erasure policies, but are not directly applicable to concrete languages. Hansen and Probst [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 [120, 121, 118, 122, 123]. Secretgrind [32], a dynamic taint tracking tool based on Valgrind [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 [19, 16, 17, 18, 15]. To remedy it, a notion of information flow-preserving program transformation has been proposed [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

[1] Bowen Alpern and Fred B. Schneider. 1987. Recognizing safety and liveness. *Distributed Comput.*, 2, 3. DOI: 10.1007/BF01782772.

[2] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*.

[3] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55, 3. DOI: 10.1145/2093548.2093564.

[4] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-c: A software analysis perspective. *Formal Aspects Comput.*, 27, 3. DOI: 10.1007/s00165-014-0326-7.

[5] Klaus Havelund and Thomas Pressburger. 2000. Model checking JAVA programs using JAVA pathfinder. *Int. J. Softw. Tools Technol. Transf.*, 2, 4. DOI: 10.1007/s100090050043.

[6] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A formally-verified C static analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. DOI: 10.1145/2676726.2676966.

[7] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The astreé analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. DOI: 10.1007/978-3-540-31987-0_3.

[8] Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, Alexandre Rebert, and Ned Williamson. 2018. The mayhem cyber reasoning system. *IEEE Secur. Priv.*, 16, 2. DOI: 10.1109/MSP.2018.1870873.

[9] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*. DOI: 10.1109/CSF.2008.7.

[10] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. DOI: 10.1145/2660267.2660283.

[11] [n. d.] BearSSL - Constant-Time Crypto. Retrieved 05/07/2019 from https://bearssl.org/constanttime.html.

[12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*. DOI: 10.1007/978-3-642-33481-8_9.

[13] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. DOI: 10.1145/3133956.3134043.

[14] Stephen Chong and Andrew C. Myers. 2005. Language-based information erasure. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. DOI: 10.1109/CSFW.2005.19.

[15] Laurent Simon, David Chisnall, and Ross J. Anderson. 2018. What you get is what you C: controlling side effects in mainstream C compilers. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. DOI: 10.1109/EuroSP.2018.00009.

[16] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. 2015. The correctness-security gap in compiler optimization. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*. DOI: 10.1109/SPW.2015.33.

[17] Frédéric Besson, Alexandre Dang, and Thomas P. Jensen. 2019. Information-flow preservation in compiler optimisations. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. DOI: 10.1109/CSF.2019.00023.

[18] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. Dead store elimination (still) considered harmful. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*.

[19] [n. d.] CWE-14: Compiler Removal of Code to Clear Buffers. Retrieved 09/29/2020 from https://cwe.mitre.org/data/definitions/14.html.

[20] J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. 2013. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming*, 78, 7. DOI: 10.1016/j.scico.2011.10.008.

[21] Sandrine Blazy, David Pichardie, and Alix Trieu. 2017. Verifying constant-time implementations by abstract interpretation. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*. DOI: 10.1007/978-3-319-66402-6_16.

[22] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*.

[23] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. 2019. Casym: cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. DOI: 10.1109/SP.2019.00022.

[24] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. 2016. When constant-time source yields variable-time binary: exploiting curve25519-donna built with MSVC 2015. In *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*. DOI: 10.1007/978-3-319-48965-0_36.

[25] Adam Langley. 2010. ImperialViolet - Checking that functions are constant time with Valgrind. Retrieved 03/14/2019 from https://www.imperialviolet.org/2010/04/01/ctgrind.html.

[26] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. 2017. Quantifying the information leak in cache attacks via symbolic execution. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*. DOI: 10.1145/3127041.3127044.

[27] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. Cached: identifying cache-based timing channels in production software. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*.

[28] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. DOI: 10.1145/3274694.3274741.

[29] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic quantification of cache side-channels. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. DOI: 10.1007/978-3-642-31424-7_40.

[30] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. Cacheaudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*.

[31] Goran Doychev and Boris Köpf. 2017. Rigorous analysis of software countermeasures against cache attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. DOI: 10.1145/3062341.3062388.

[32] 2020. Secretgrind. (22, 2020). Retrieved 09/29/2020 from https://github.com/lmrs2/secretgrind.

[33] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. DOI: 10.1109/CSFW.2004.17.

[34] Tachio Terauchi and Alexander Aiken. 2005. Secure information flow as a safety problem. In *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. DOI: 10.1007/11547662_24.

[35] Adel Djoudi, Sébastien Bardin, and Éric Goubault. 2016. Recovering high-level conditions from binary programs. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*. DOI: 10.1007/978-3-319-48989-6_15.

[36] Gogul Balakrishnan and Thomas W. Reps. 2010. WYSINWYX: what you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32, 6. DOI: 10.1145/1749608.1749612.

[37] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56, 2. DOI: 10.1145/2408776.2408795.

[38] Ella Bounimova, Patrice Godefroid, and David A. Molnar. 2013. Billions and billions of constraints: whitebox fuzz testing in production. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. DOI: 10.1109/ICSE.2013.6606558.

[39] Musard Balliu, Mads Dam, and Roberto Guanciale. 2014. Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. DOI: 10.1145/2660267.2660322.

[40] Quoc Huy Do, Richard Bubel, and Reiner Hähnle. 2015. Exploit generation for information flow leaks in object-oriented programs. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*. DOI: 10.1007/978-3-319-18467-8_27.

[41] Dimiter Milushev, Wim Beck, and Dave Clarke. 2012. Noninterference via symbolic execution. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*. DOI: 10.1007/978-3-642-30793-5_10.

[42] Pramod Subramanyan, Sharad Malik, Hareesh Khattri, Abhranil Maiti, and Jason M. Fung. 2016. Verifying information flow properties of firmware using symbolic execution. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*.

[43] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. DOI: 10.1145/964001.964003.

[44] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*. DOI: 10.1007/978-3-642-21437-0_17.

[45] Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. DOI: 10.1145/2103656.2103677.

[46] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. 2018. A better facet of dynamic information flow control. In *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*. DOI: 10.1145/3184558.3185979.

[47] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. DOI: 10.1145/2884781.2884845.

[48] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. 2019. Relational symbolic execution. In *PPDP*.

[49] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2022. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *CoRR*, abs/2209.01129. DOI: 10.48550/ARXIV.2209.01129.

[50] Nadhem J. AlFardan and Kenneth G. Paterson. 2013. Lucky thirteen: breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. DOI: 10.1109/SP.2013.42.

[51] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/rel: efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. DOI: 10.1109/SP40000.2020.00074.

[52] Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Commun. ACM*, 20, 7. DOI: 10.1145/359636.359712.

[53] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding data lifetime via whole system simulation (awarded best paper!) In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*.

[54] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30, 1. DOI: 10.1145/2110356.2110358.

[55] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (state of) the art of war: offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. DOI: 10.1109/SP.2016.17.

[56] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. DOI: 10.1109/SANER.2016.43.

[57] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM*, 19, 7. DOI: 10.1145/360248.360252.

[58] Julien Vanegue and Sean Heelan. 2012. SMT solvers in software security. In *6th USENIX Workshop on Offensive Technologies, WOOT'12, August 6-7, 2012, Bellevue, WA, USA, Proceedings*.

[59] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*.

[60] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: exploit hardening made easy. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*.

[61] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. DOI: 10.1109/SP.2015.47.

[62] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-bounded DSE: targeting infeasibility questions on obfuscated codes. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. DOI: 10.1109/SP.2017.36.

[63] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. 2018. Symbolic deobfuscation: from virtualized code back to the original. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*. DOI: 10.1007/978-3-319-93411-2_17.

[64] Adel Djoudi and Sébastien Bardin. 2015. BINSEC: binary code analysis with low-level regions. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part*

of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. DOI: 10.1007/978-3-662-46681-0_17.

[65] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A binary analysis platform. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings.* DOI: 10.1007/978-3-642-22110-1_37.

[66] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. The SMT-LIB Standard: Version 2.6.

[67] [n. d.] FixedSizeBitVectors Theory, SMT-LIB. Retrieved 04/02/2019 from http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml.

[68] [n. d.] ArraysEx Theory, SMT-LIB. Retrieved 04/02/2019 from http://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml.

[69] Quoc-Sang Phan. 2013. Self-composition by symbolic execution. In *2013 Imperial College Computing Student Workshop, ICCSW 2013, September 26/27, 2013, London, United Kingdom.* DOI: 10.4230/OASIcs.ICCSW.2013.95.

[70] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* DOI: 10.1007/978-3-540-78800-3_24.

[71] Daniel J. Bernstein. 2006. Curve25519: new diffie-hellman speed records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings.* DOI: 10.1007/11745853_14.

[72] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. 2011. The BINCOA framework for binary code analysis. In *CAV*.

[73] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018.* DOI: 10.1109/CSF.2018.00031.

[74] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. 2018. Arrays made simpler: an efficient, scalable and thorough preprocessing. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018.* DOI: 10.29007/dc9b.

[75] Greg Nelson and Derek C. Oppen. 1979. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1, 2. DOI: 10.1145/357073.357079.

[76] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9.

[77] [n. d.] SMT-COMP. SMT-COMP. Retrieved 10/11/2019 from https://smt-comp.github.io/2019/results.html.

[78] [n. d.] Imdea-software/verifying-constant-time. GitHub. Retrieved 10/13/2019 from https://github.com/imdea-software/verifying-constant-time.

[79] [n. d.] OpenSSL, Cryptography and SSL/TLS Toolkit. Retrieved 04/24/2021 from https://www.openssl.org/.

[80] David J. Wheeler and Roger M. Needham. 1994. Tea, a tiny encryption algorithm. In *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings.* DOI: 10.1007/3-540-60590-8_29.

[81] Thomas Pornin. [n. d.] BearSSL. Retrieved 05/23/2019 from https://www.bearssl.org/.

[82] Daan Sprenkels. [n. d.] LLVM provides no side-channel resistance. Retrieved 10/01/2021 from https://dsprenkels.com/cmov-conversion.html.

[83] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: automatic side-channel resistance using efficient control and data flow linearization. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. DOI: 10.1145/3460120.3484583.

[84] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*.

[85] [n. d.] Openssl, `openssl_cleanse` function. Retrieved 04/24/2021 from https://github.com/openssl/openssl/blob/master/crypto/mem_clr.c.

[86] [n. d.] Libgcrypt, `wipememory` function. Retrieved 04/24/2021 from https://github.com/equalitie/libgcrypt/blob/libgcrypt-1.6.3/src/gcryptrnd.c.

[87] [n. d.] wolfSSL, `ForceZero` function. Retrieved 04/24/2021 from https://github.com/equalitie/libgcrypt/blob/libgcrypt-1.6.3/src/gcryptrnd.c.

[88] Todd C. Miller. [n. d.] sudo, `explicit_bzero` function. Retrieved 04/24/2021 from https://github.com/sudo-project/sudo/blob/SUDO_1_9_6/lib/util/explicit_bzero.c.

[89] [n. d.] libsodium, `sodium_memzero` function. Retrieved 04/24/2021 from https://github.com/jedisct1/libsodium/blob/1.0.18/src/libsodium/sodium/utils.c.

[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.

[91] Reini Urban. [n. d.] Safeclib, `MEMORY_BARRIER` macro. Retrieved 04/24/2021 from https://github.com/rurban/safeclib/blob/v31082020/src/mem/mem_primitives_lib.h.

[92] [n. d.] 6.47.2 Extended Asm - Assembler Instructions with C Expression Operands. Retrieved 04/24/2021 from https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html.

[93] [n. d.] Bug 15495 - dead store pass ignores memory clobbering asm statement. Retrieved 04/24/2021 from https://bugs.llvm.org/show_bug.cgi?id=15495.

[94] Reini Urban. [n. d.] Safeclib, `memset_s` function. Retrieved 04/24/2021 from https://github.com/rurban/safeclib/blob/v31082020/src/mem/memset_s.c.

[95] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. DOI: 10.1109/SP.2019.00002.

[96] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-Laure Potet. 2019. Get rid of inline assembly through verification-oriented lifting. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. DOI: 10.1109/ASE.2019.00060.

[97] Benjamin Farinier, Sébastien Bardin, Richard Bonichon, and Marie-Laure Potet. 2018. Model generation for quantified formulas: A taint-based approach. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. DOI: 10.1007/978-3-319-96142-2_19.

[98] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. 2016. Specification of concretization and symbolization policies in symbolic execution. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. DOI: 10.1145/2931037.2931048.

[99] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungil Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing intermediate representations for binary analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. DOI: 10.1109/ASE.2017.8115648.

[100] Musard Balliu, Mads Dam, and Gurvan Le Guernic. 2012. Encover: symbolic exploration for information flow security. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. DOI: 10.1109/CSF.2012.24.

[101] Cristian Cadar and Hristina Palikareva. 2014. Shadow symbolic execution for better testing of evolving software. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*. DOI: 10.1145/2591062.2591104.

[102] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. 2017. Dude, is my code constant time? In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. DOI: 10.23919/DATE.2017.7927267.

[103] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. 2020. Ct-fuzz: fuzzing for timing leaks. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. DOI: 10.1109/ICST46399.2020.00063.

[104] Johan Agat. 2000. Transforming out timing leaks. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*. DOI: 10.1145/325694.325702.

[105] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The program counter security model: automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*. DOI: 10.1007/11734727_14.

[106] Bruno Rodrigues, Fernando Magno Quint
textasciitilde ao Pereira, and Diego F. Aranha. 2016. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. DOI: 10.1145/2892208.2892230.

[107] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: high-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. DOI: 10.1145/3133956.3134078.

[108] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. 2017. Vale: verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*.

[109] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. Fact: A flexible, constant-time programming language. In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*. DOI: 10.1109/SecDev.2017.24.

[110] Boris Köpf and Heiko Mantel. 2007. Transformational typing and unification for automatically correcting insecure programs. *Int. J. Inf. Sec.*, 6, 2-3. DOI: 10.1007/s10207-007-0016-z.

[111] Sudipta Chattopadhyay and Abhik Roychoudhury. 2018. Symbolic verification of cache side-channel freedom. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 37, 11. DOI: 10.1109/TCAD.2018.2858402.

[112] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. DOI: 10.1145/3213846.3213851.

[113] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. DOI: 10.1109/SP.2009.19.

[114] Luigi Soares and Fernando Magno Quint
textasciitilde ao Pereira. 2021. Memory-safe elimination of side channels. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*. DOI: 10.1109/CGO51591.2021.9370305.

[115] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. 2005. Shredding your garbage: reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*.

[116] Tal Garfinkel, Ben Pfaff, Jim Chow, and Mendel Rosenblum. 2004. Data lifetime is a systems problem. In *Proceedings of the 11st ACM SIGOPS European Workshop, Leuven, Belgium, September 19-22, 2004*. DOI: 10.1145/1133572.1133599.

[117] Filippo Del Tedesco, Sebastian Hunt, and David Sands. 2011. A semantic hierarchy for erasure policies. In *Information Systems Security - 7th International Conference, ICISS 2011, Kolkata, India, December 15-19, 2011, Procedings*. DOI: 10.1007/978-3-642-25560-1_24.

[118] Aslan Askarov, Scott Moore, Christos Dimoulas, and Stephen Chong. 2015. Cryptographic enforcement of language-based information erasure. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. DOI: 10.1109/CSF.2015.30.

[119] René Rydhof Hansen and Christian W Probst. 2006. Non-interference and erasure policies for java card bytecode. In *6th International Workshop on Issues in the Theory of Security (WITS'06)*.

[120] Sebastian Hunt and David Sands. 2008. Just forget it - the semantics and enforcement of information erasure. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. DOI: 10.1007/978-3-540-78739-6_19.

[121] Stephen Chong and Andrew C. Myers. 2008. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*. DOI: 10.1109/CSF.2008.12.

[122] Filippo Del Tedesco, Alejandro Russo, and David Sands. 2010. Implementing erasure policies using taint analysis. In *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers*. DOI: 10.1007/978-3-642-27937-9_14.

[123] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2011. Verification of information flow and access control policies with dependent types. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. DOI: 10.1109/SP.2011.12.

[124] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. DOI: 10.1145/1250734.1250746.

[125] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM*, 52, 7. DOI: 10.1145/1538788.1538814.