Lazy Self-Composition for Security Verification*

Weikun Yang[1], Yakir Vizel[1], Pramod Subramanyan[2], Aarti Gupta[1], and Sharad Malik[1]

[1] Princeton University
[2] University of California, Berkeley

Abstract. The secure information flow problem, which checks whether low-security outputs of a program are influenced by high-security inputs, has many applications in verifying security properties in programs. In this paper we present *lazy* self-composition, an approach for verifying secure information flow. It is based on self-composition, where two copies of a program are created on which a safety property is checked. However, rather than an eager duplication of the given program, it uses duplication lazily to reduce the cost of verification. This lazy self-composition is guided by an interplay between symbolic taint analysis on an abstract (single copy) model and safety verification on a refined (two copy) model. We propose two verification methods based on lazy self-composition. The first is a CEGAR-style procedure, where the abstract model associated with taint analysis is refined, on demand, by using a model generated by lazy self-composition. The second is a method based on bounded model checking, where taint queries are generated dynamically during program unrolling to guide lazy self-composition and to conclude an adequate bound for correctness. We have implemented these methods on top of the SEAHORN verification platform and our evaluations show the effectiveness of lazy self-composition.

1 Introduction

Many security properties can be cast as the problem of verifying secure information flow. A standard approach to verifying secure information flow is to reduce it to a safety verification problem on a "self-composition" of the program, i.e., two "copies" of the program are created [5] and analyzed. For example, to check for information leaks or non-interference [16], low-security (public) inputs are initialized to identical values in the two copies of the program, while high-security (confidential) inputs are unconstrained and can take different values. The safety check ensures that in all executions of the two-copy program, the values of the low-security (public) outputs are identical, i.e., there is no information leak from confidential inputs to public outputs. The self-composition approach is useful for checking general hyper-properties [10], and has been used in other applications, such as verifying constant-time code for security [1] and k-safety properties of functions like injectivity and monotonicity [32].

Although the self-composition reduction is sound and complete, it is challenging in practice to check safety properties on two copies of a program. There have been many efforts to reduce the cost of verification on self-composed programs, e.g., by use of type-based analysis [33], constructing product programs with aligned fragments [4], lockstep execution of loops [32], transforming Horn clause rules [13,24], etc. The underlying

* This work was supported in part by NSF Grant 1525936.

theme in these efforts is to make it easier to derive *relational* invariants between the two copies, e.g., by keeping corresponding variables in the two copies near each other.

In this paper, we aim to improve the self-composition approach by making it *lazier* in contrast to eager duplication into two copies of a program. Specifically, we use symbolic taint analysis to track flow of information from high-security inputs to other program variables. (This is similar to dynamic taint analysis [30], but covers all possible inputs due to static verification.) This analysis works on an abstract model of a single copy of the program and employs standard model checking techniques for achieving precision and path sensitivity. When this abstraction shows a counterexample, we refine it using on-demand duplication of relevant parts of the program. Thus, our *lazy self-composition*[3] approach is guided by an interplay between symbolic taint analysis on an abstract (single copy) model and safety verification on a refined (two copy) model.

We describe two distinct verification methods based on lazy self-composition. The first is an iterative procedure for unbounded verification based on counterexample guided abstraction refinement (CEGAR) [8]. Here, the taint analysis provides a sound over-approximation for secure information flow, i.e., if a low-security output is proved to be untainted, then it is guaranteed to not leak any information. However, even a path-sensitive taint analysis can sometimes lead to "false alarms", i.e., a low-security output is tainted, but its value is unaffected by high-security inputs. For example, this can occur when a branch depends on a tainted variable, but the same (semantic, and not necessarily syntactic) value is assigned to a low-security output on both branches. Such false alarms for security due to taint analysis are then refined by lazily duplicating relevant parts of a program, and performing a safety check on the composed two-copy program. Furthermore, we use relational invariants derived on the latter to strengthen the abstraction within the iterative procedure.

Our second method also takes a similar abstraction-refinement view, but in the framework of bounded model checking (BMC) [6]. Here, we dynamically generate taint queries (in the abstract single copy model) during program unrolling, and use their result to simplify the duplication for self-composition (in the two copy model). Specifically, the second copy duplicates the statements (update logic) only if the taint query shows that the updated variable is possibly tainted. Furthermore, we propose a specialized early termination check for the BMC-based method. In many secure programs, sensitive information is propagated in a localized context, but conditions exist that squash its propagation any further. We formulate the early termination check as a taint check on all live variables at the end of a loop body, i.e., if no live variable is tainted, then we can conclude that the program is secure without further loop unrolling. (This is under the standard assumption that inputs are tainted in the initial state. The early termination check can be suitably modified if tainted inputs are allowed to occur later.) Since our taint analysis is precise and path-sensitive, this approach can be beneficial in practice by unrolling the loops past the point where all taint has been squashed.

We have implemented these methods in the SEAHORN verification platform [17], which represents programs as CHC (Constrained Horn Clause) rules. Our prototype for taint analysis is flexible, with a fully symbolic encoding of the taint policy (i.e., rules for taint generation, propagation, and removal). It fully leverages SMT-based model

[3] This name is inspired by the *lazy abstraction* approach [19] for software model checking.

checking techniques for precise taint analysis. Our prototypes allow rich security specifications in terms of annotations on low/high-security variables and locations in arrays, and predicates that allow information downgrading in specified contexts.

We present an experimental evaluation on benchmark examples. Our results clearly show the benefits of lazy self-composition vs. eager self-composition, where the former is much faster and allows verification to complete in larger examples. Our initial motivation in proposing the two verification methods was that we would find examples where one or the other method is better. We expect that easier proofs are likely to be found by the CEGAR-based method, and easier bugs by the BMC-based method. As it turns out, most of our benchmark examples are easy to handle by both methods so far. We believe that our general approach of lazy self-composition would be beneficial in other verification methods, and both our methods show its effectiveness in practice.

To summarize, this paper makes the following contributions.

- We present lazy self-composition, an approach to verifying secure information flow that reduces verification cost by exploiting the interplay between a path-sensitive symbolic taint analysis and safety checking on a self-composed program.
- We present IFC-CEGAR, a procedure for unbounded verification of secure information flow based on lazy self-composition using the CEGAR paradigm. IFC-CEGAR starts with a taint analysis abstraction of information flow and iteratively refines this abstraction using self-composition. It is tailored toward proving that programs have secure information flow.
- We present IFC-BMC, a procedure for bounded verification of secure information flow. As the program is being unrolled, IFC-BMC uses dynamic symbolic taint checks to determine which parts of the program need to be duplicated. This method is tailored toward bug-finding.
- We develop prototype implementations of IFC-CEGAR and IFC-BMC and present an experimental evaluation of these methods on a set of benchmarks/microbenchmarks. Our results demonstrate that IFC-CEGAR and IFC-BMC easily outperform an eager self-composition that uses the same backend verification engines.

2 Motivating Example

Listing 1 shows a snippet from a function that performs multiword multiplication. The code snippet is instrumented to count the number of iterations of the inner loop that are executed in `bigint_shiftleft` and `bigint_add` (not shown for brevity). These iterations are counted in the variable `steps`. The security requirement is that `steps` must not depend on the secret values in the array `a`; array `b` is assumed to be public.

Static analyses, including those based on security types, will conclude that the variable `steps` is “high-security.” This is because `steps` is assigned in a conditional branch that depends on the high-security variable `bi`. However, this code is in fact safe because steps is incremented by the same value in both branches of the conditional statement.

Our lazy self-composition will handle this example by first using a symbolic taint analysis to conclude that the variable `steps` is tainted. It will then self-compose only those parts of the program related to computation of `steps`, and discover that it is set to identical values in both copies, thus proving the program is secure.

```
int steps = 0;
for (i = 0; i < N; i++) { zero[i] = product[i] = 0; }
for (i = 0; i < N*W; i++) {
  int bi = bigint_extract_bit(a, i);
  if (bi == 1) {
    bigint_shiftleft(b, i, shifted_b, &steps);
    bigint_add(product, shifted_b, product, &steps);
  } else {
    bigint_shiftleft(zero, i, shifted_zero, &steps);
    bigint_add(product, shifted_zero, product, &steps);
  }
}
```

Listing 1: “BigInt” Multiplication

Now consider the case when the code in Listing 1 is used to multiply two “big-ints” of differing widths, e.g., a 512b integer is multiplied with 2048b integer. If this occurs, the upper 1536 bits of `a` will all be zeros, and `bi` will not be a high-security variable for these iterations of the loop. Such a scenario can benefit from early-termination in our BMC-based method: our analysis will determine that no tainted value flows to the low security variable `steps` after iteration 512 and will immediately terminate the analysis.

3 Preliminaries

We consider First Order Logic modulo a theory $\mathcal{T}$ and denote it by $FOL(\mathcal{T})$. Given a program P, we define a *safety verification* problem w.r.t. P as a transition system $M = \langle X, \mathit{Init}(X), \mathit{Tr}(X, X'), \mathit{Bad}(X)\rangle$ where X denotes a set of (uninterpreted) constants, representing program variables; Init, Tr and Bad are (quantifier-free) formulas in $FOL(\mathcal{T})$ representing the initial states, transition relation and bad states, respectively. The states of a transition system correspond to structures over a signature $\Sigma = \Sigma_{\mathcal{T}} \cup X$. We write $\mathit{Tr}(X, X')$ to denote that Tr is defined over the signature $\Sigma_{\mathcal{T}} \cup X \cup X'$, where X is used to represent the pre-state of a transition, and $X' = \{a' \mid a \in X\}$ is used to represent the post-state.

A safety verification problem is to decide whether a transition system M is SAFE or UNSAFE. We say that M is UNSAFE iff there exists a number N such that the following formula is satisfiable:

$$\mathit{Init}(X_0) \wedge \left(\bigwedge_{i=0}^{N-1} \mathit{Tr}(X_i, X_{i+1})\right) \wedge \mathit{Bad}(X_N) \tag{1}$$

where $X_i = \{a_i \mid a \in X\}$ is a copy of the program variables (uninterpreted constants) used to represent the state of the system after the execution of i steps.

When M is UNSAFE and $s_N \in \mathit{Bad}$ is reachable, the path from $s_0 \in \mathit{Init}$ to s_N is called a *counterexample* (CEX).

A transition system M is SAFE iff the transition system has no counterexample, of any length. Equivalently, M is SAFE iff there exists a formula Inv, called a *safe inductive invariant*, that satisfies: (i) $\mathit{Init}(X) \to \mathit{Inv}(X)$, (ii) $\mathit{Inv}(X) \wedge \mathit{Tr}(X, X') \to \mathit{Inv}(X')$, and (iii) $\mathit{Inv}(X) \to \neg \mathit{Bad}(X)$.

In SAT-based model checking (e.g., based on IC3 [?] or interpolants [?,?]), the verification procedure maintains an *inductive trace* of formulas $[F_0(X), \ldots, F_N(X)]$ that satisfy: (i) $\mathit{Init}(X) \rightarrow F_0(X)$, (ii) $F_i(X) \wedge \mathit{Tr}(X, X') \rightarrow F_{i+1}(X')$ for every $0 \leq i < N$, and (iii) $F_i(X) \rightarrow \neg \mathit{Bad}(X)$ for every $0 \leq i \leq N$. A trace $[F_0, \ldots, F_N]$ is *closed* if $\exists 1 \leq i \leq N \cdot F_i \Rightarrow \left(\bigvee_{j=0}^{i-1} F_j\right)$. There is an obvious relationship between existence of closed traces and safety of a transition system: *A transition system T is SAFE iff it admits a safe closed trace.* Thus, safety verification is reduced to searching for a safe closed trace or finding a CEX.

4 Information Flow Analysis

Let P be a program over a set of program variables X. Recall that $\mathit{Init}(X)$ is a formula describing the initial states and $\mathit{Tr}(X, X')$ a transition relation. We assume a "stuttering" transition relation, namely, Tr is reflexive and therefore it can non-deterministically either move to the next state or stay in the same state. Let us assume that $H \subset X$ is a set of high-security variables and $L := X \backslash H$ is a set of low-security variables.

For each $x \in L$, let $\mathit{Obs}_x(X)$ be a predicate over program variables X that determines when variable x is adversary-observable. The precise definition of $\mathit{Obs}_x(X)$ depends on the threat model being considered. A simple model would be that for each low variable $x \in L$, $\mathit{Obs}_x(X)$ holds only at program completion – this corresponds to a threat model where the adversary can run a program that operates on some confidential data and observe its public (low-security) outputs after completion. A more sophisticated definition of $\mathit{Obs}_x(X)$ could consider, for example, a concurrently executing adversary. Appropriate definitions of $\mathit{Obs}_x(X)$ can also model declassification [29], by setting $\mathit{Obs}_x(X)$ to be false in program states where the declassification of x is allowed.

The *information flow* problem checks whether there exists an execution of P such that the value of variables in H affects a variable in $x \in L$ in some state where the predicate $\mathit{Obs}_x(X)$ holds. Intuitively, information flow analysis checks if low-security variables "leak" information about high-security variables.

We now describe our formulations of two standard techniques that have been used to perform information flow analysis. The first is based on taint analysis [30], but we use a symbolic (rather than a dynamic) analysis that tracks taint in a path-sensitive manner over the program. The second is based on self-composition [5], where two copies of the program are created and a safety property is checked over the composed program.

4.1 Symbolic Taint Analysis

When using taint analysis for checking information flow, we mark high-security variables with a "taint" and check if this taint can propagate to low-security variables. The propagation of taint through program variables of P is determined by both assignments and the control structure of P. In order to perform precise taint analysis, we formulate it as a safety verification problem. For this purpose, for each program variable $x \in X$, we introduce a new "taint" variable x_t. Let $X_t := \{x_t \mid x \in X\}$ be the set of taint variables where $x_t \in X_t$ is of sort Boolean. Let us define a transition system $M_t := \langle Y, \mathit{Init}_t, \mathit{Tr}_t, \mathit{Bad}_t \rangle$ where $Y := X \cup X_t$ and

$$\mathit{Init}_t(Y) := \mathit{Init}(X) \wedge \left(\bigwedge_{x \in H} x_t \right) \wedge \left(\bigwedge_{x \in L} \neg x_t \right) \tag{2}$$

$$\mathit{Tr}_t(Y, Y') := \mathit{Tr}(X, X') \wedge \hat{\mathit{Tr}}(Y, X'_t) \tag{3}$$

$$\mathit{Bad}_t(Y) := \left(\bigvee_{x \in L} \mathit{Obs}_x(X) \wedge x_t \right) \tag{4}$$

Since taint analysis tracks information flow from high-security to low-security variables, variables in H_t are initialized to true while variables in L_t are initialized to false. W.l.o.g., let us denote the state update for a program variable $x \in X$ as: $x' = \mathit{cond}(X)\ ?\ \varphi_1(X)\ :\ \varphi_2(X)$. Let φ be a formula over Σ. We capture the taint of φ by:

$$\Theta(\varphi) = \begin{cases} \mathit{false} & \text{if } \varphi \cap X = \emptyset \\ \bigvee\limits_{x \in \varphi} x_t & \text{otherwise} \end{cases}$$

Thus, $\hat{\mathit{Tr}}(X_t, X'_t)$ is defined as: $\bigwedge\limits_{x_t \in X_t} x'_t = \Theta(\mathit{cond}) \vee (\mathit{cond}\ ?\ \Theta(\varphi_1)\ :\ \Theta(\varphi_2))$

Intuitively, taint may propagate from x_1 to x_2 either when x_1 is assigned an expression that involves x_2 or when an assignment to x_1 is controlled by x_2. The bad states (Bad_t) are all states where a low-security variable is tainted and observable.

4.2 Self-Composition

When using self-composition, information flow is tracked over an execution of two copies of the program, P and P_d. Let us denote $X_d := \{x_d \mid x \in X\}$ as the set of program variables of P_d. Similarly, let $\mathit{Init}_d(X_d)$ and $\mathit{Tr}_d(X_d, X'_d)$ denote the initial states and transition relation of P_d. Note that Init_d and Tr_d are computed from Init and Tr by means of substitutions. Namely, substituting every occurrence of $x \in X$ or $x' \in X'$ with $x_d \in X_d$ and $x'_d \in X'_d$, respectively. Similarly to taint analysis, we formulate information flow over a self-composed program as a safety verification problem: $M_d := \langle Z, \mathit{Init}_d, \mathit{Tr}_d, \mathit{Bad}_d \rangle$ where $Z := X \cup X_d$ and

$$\mathit{Init}_d(Z) := \mathit{Init}(X) \wedge \mathit{Init}(X_d) \wedge \left(\bigwedge_{x \in L} x = x_d \right) \tag{5}$$

$$\mathit{Tr}_d(Z, Z') := \mathit{Tr}(X, X') \wedge \mathit{Tr}(X_d, X'_d) \tag{6}$$

$$\mathit{Bad}_d(Z) := \left(\bigvee_{x \in L} \mathit{Obs}_x(X) \wedge \mathit{Obs}_x(X_d) \wedge \neg(x = x_d) \right) \tag{7}$$

In order to track information flow, variables in L_d are initialized to be equal to their counterpart in L, while variables in H_d remain unconstrained. A leak is captured by the bad states (i.e. Bad_d). More precisely, there exists a leak iff there exists an execution of M_d that results in a state where $\mathit{Obs}_x(X)$, $\mathit{Obs}_x(X_d)$ hold and $x \neq x_d$ for a low-security variable $x \in L$.

5 Lazy Self-Composition for Information Flow Analysis

In this section, we introduce lazy self-composition for information flow analysis. It is based on an interplay between symbolic taint analysis on a single copy and safety verification on a self-composition, which were both described in the previous section.

Recall that taint analysis is imprecise for determining secure information flow in the sense that it may report spurious counterexamples, namely, spurious leaks. In contrast, self-composition is precise, but less efficient. The fact that self composition requires a duplication of the program often hinders its performance. The main motivation for lazy self-composition is to target both efficiency and precision.

Intuitively, the model for symbolic taint analysis M_t can be viewed as an abstraction of the self-composed model M_d, where the Boolean variables in M_t are predicates tracking the states where $x \neq x_d$ for some $x \in X$. This intuition is captured by the following statement: *M_t over-approximates M_d.*

Corollary 1. *If there exists a path in M_d from $Init_d$ to Bad_d then there exists a path in M_t from $Init_t$ to Bad_t.*

Corollary 2. *If there exists no path in M_t from $Init_t$ to Bad_t then there exists no path in M_d from $Init_d$ to Bad_d.*

This abstraction-based view relating symbolic taint analysis and self-composition can be exploited in different verification methods for checking secure information flow. In this paper, we focus on two – a CEGAR-based method (IFC-CEGAR) and a BMC-based method (IFC-BMC). These methods using lazy self-composition are now described in detail.

5.1 IFC-CEGAR

We make use of the fact that M_t can be viewed as an abstraction w.r.t. to M_d, and propose an abstraction-refinement paradigm for secure information flow analysis. In this setting, M_t is used to find a possible counterexample, i.e., a path that leaks information. Then, M_d is used to check if this counterexample is spurious or real. In case the counterexample is found to be spurious, IFC-CEGAR uses the proof that shows why the counterexample is not possible in M_d to refine M_t.

A sketch of IFC-CEGAR appears in Alg 1. Recall that we assume that solving a safety verification problem is done by maintaining an inductive trace. We denote the traces for M_t and M_d by $\boldsymbol{G} = [G_0, \ldots, G_k]$ and $\boldsymbol{H} = [H_0, \ldots, H_k]$, respectively. IFC-CEGAR starts by initializing M_t, M_d and their respective traces $\boldsymbol{G}$ and $\boldsymbol{H}$ (lines 1-4). The main loop of IFC-CEGAR (lines 5-18) starts by looking for a counterexample over M_t (line 6). In case no counterexample is found, IFC-CEGAR declares there are no leaks and returns SAFE.

If a counterexample π is found in M_t, IFC-CEGAR first updates the trace of M_d, i.e. $\boldsymbol{H}$, by rewriting $\boldsymbol{G}$ (line 10). In order to check if π is spurious, IFC-CEGAR creates a new safety verification problem M_c, a version of M_d constrained by π (line 11) and solves it (line 12). If M_c has a counterexample, IFC-CEGAR returns UNSAFE.

Algorithm 1: IFC-CEGAR (P,H)

```
Input: A program $P$ and a set of high-security variables $H$
Output: SAFE, UNSAFE or UNKNOWN.
1  $M_t \leftarrow$ ConstructTaintModel($P, H$)
2  $M_d \leftarrow$ ConstructSCModel($P, H$)
3  $\boldsymbol{G} \leftarrow [G_0 = Init_t]$
4  $\boldsymbol{H} \leftarrow [H_0 = Init_d]$
5  repeat
6      $(\boldsymbol{G}, R_{taint}, \pi) \leftarrow$ MC.Solve($M_t, \boldsymbol{G}$)
7      if $R_{taint} = SAFE$ then
8          return SAFE
9      else
10         $\boldsymbol{H} \leftarrow$ ReWrite($\boldsymbol{G}, \boldsymbol{H}$)
11         $M_c \leftarrow$ Constraint($M_d, \pi$)
12         $(\boldsymbol{H}, R_s, \pi) \leftarrow$ MC.Solve($M_c, \boldsymbol{H}$)
13         if $R_s = UNSAFE$ then
14             return UNSAFE
15         else
16             $\boldsymbol{G} \leftarrow$ ReWrite($\boldsymbol{H}, \boldsymbol{G}$)
17             $M_t \leftarrow$ Refine($M_t, \boldsymbol{G}$)
18 until $\infty$
19 return UNKNOWN
```

Otherwise, $\boldsymbol{G}$ is updated by $\boldsymbol{H}$ (line 16) and M_t is refined such that π is ruled out (line 17).

The above gives a high-level overview of how IFC-CEGAR operates. We now go into more detail. More specifically, we describe the functions `ReWrite`, `Constraint` and `Refine`. We note that these functions can be designed and implemented in several different ways. In what follows we describe some possible choices.

Proof-based Abstraction Let us assume that when solving M_t a counterexample π of length k is found and an inductive trace $\boldsymbol{G}$ is computed. Following a proof-based abstraction approach, `Constraint()` uses the length of π to bound the length of possible executions in M_d by k. Intuitively, this is similar to bounding the length of the computed inductive trace over M_d.

In case M_c has a counterexample, a real leak (of length k) is found. Otherwise, since M_c considers all possible executions of M_d of length k, IFC-CEGAR deduces that there are no counterexamples of length k. In particular, the counterexample π is ruled out. IFC-CEGAR therefore uses this fact to refine M_t and $\boldsymbol{G}$.

Inductive Trace Rewriting Consider the set of program variables X, taint variables X_t, and self compositions variables X_d. As noted above, M_t over-approximates M_d. Intuitively, it may mark a variable x as tainted when x does not leak information. Equivalently, if a variable x is found to be untainted in M_t then it is known to also not leak information in M_d. More formally, the following relation holds: $\neg x_t \rightarrow (x = x_d)$.

This gives us a procedure for rewriting a trace over M_t to a trace over M_d. Let $\boldsymbol{G} = [G_0, \dots, G_k]$ be an inductive trace over M_t. Considering the definition of M_t, $\boldsymbol{G}$ can be decomposed and rewritten as: $G_i(Y) := \bar{G}_i(X) \land \bar{G}_i^t(X_t) \land \psi(X, X_t)$. Namely, $\bar{G}_i(X)$ and $\bar{G}_i^t(X_t)$ are sub-formulas of G_i over only X and X_t variables, respectively, and $\psi(X, X_t)$ is the part connecting X and X_t.

Since $\boldsymbol{G}$ is an inductive trace $G_i(Y) \land Tr_t(Y, Y') \to G_{i+1}(Y')$ holds. Following the definition of Tr_t and the above decomposition of G_i, the following holds:

$$\bar{G}_i(X) \land Tr(X, X') \to \bar{G}_{i+1}(X')$$

Let $\boldsymbol{H} = [H_0, \dots, H_k]$ be a trace w.r.t. M_d. We define the *update* of $\boldsymbol{H}$ by $\boldsymbol{G}$ as the trace $\boldsymbol{H}^* = [H_0^*, \dots, H_k^*]$, which is defined as follows:

$$H_0^* := Init_d \tag{8}$$

$$H_i^*(Z) := H_i(Z) \land \bar{G}_i(X) \land \bar{G}_i(X_d) \land \Big(\bigwedge \{x = x_d \mid G_i(Y) \to \neg x_t\}\Big) \tag{9}$$

Intuitively, if a variable $x \in X$ is known to be untainted in M_t, using Corollary 2 we conclude that $x = x_d$ in M_d.

A similar update can be defined when updating a trace $\boldsymbol{G}$ w.r.t. M_t by a trace $\boldsymbol{H}$ w.r.t. M_d. In this case, we use the following relation: $\neg(x = x_d) \to x_t$. Let $\boldsymbol{H} = [H_0(Z), \dots, H_k(Z)]$ be the inductive trace w.r.t. M_d. $\boldsymbol{H}$ can be decomposed and written as $H_i(Z) := \bar{H}_i(X) \land \bar{H}_i^d(X_d) \land \phi(X, X_d)$.

Due to the definition of M_d and an inductive trace, the following holds:

$$\bar{H}_i(X) \land Tr(X, X') \to \bar{H}_i(X')$$

$$\bar{H}_i^d(X_d) \land Tr(X_d, X_d') \to \bar{H}_i^d(X_d')$$

We can therefore update a trace $\boldsymbol{G} = [G_0, \dots, G_k]$ w.r.t. M_t by defining the trace $\boldsymbol{G}^* = [G_0^*, \dots, G_k^*]$, where:

$$G_0^* := Init_d \tag{10}$$

$$G_i^*(Y) := G_i(Y) \land \bar{H}_i(X) \land \bar{H}_i^d(X) \land \Big(\bigwedge \{x_t \mid H_i(Z) \to \neg(x = x_d)\}\Big) \tag{11}$$

Updating $\boldsymbol{G}$ by $\boldsymbol{H}$, and vice-versa, as described above is based on the fact that M_t over-approximates M_d w.r.t. tainted variables (namely, Corollary 1 and 2). It is therefore important to note that $\boldsymbol{G}^*$ in particular, does not "gain" more precision due to this process.

Lemma 1. *Let $\boldsymbol{G}$ be an inductive trace w.r.t. M_t and $\boldsymbol{H}$ an inductive trace w.r.t. M_d. Then, the updated $\boldsymbol{H}^*$ and $\boldsymbol{G}^*$ are inductive traces w.r.t. M_d and M_t, respectively.*

Refinement Recall that in the current scenario, a counterexample was found in M_t, and was shown to be spurious in M_d. This fact can be used to refine both M_t and $\boldsymbol{G}$.

As a first step, we observe that if $x = x_d$ in M_d, then $\neg x_t$ should hold in M_t. However, since M_t is an over-approximation it may allow x to be tainted, namely, allow x_t to be evaluated to $true$.

In order to refine M_t and $\boldsymbol{G}$, we define a strengthening procedure for $\boldsymbol{G}$, which resembles the updating procedure that appears in the previous section. Let $\boldsymbol{H} = [H_0, \dots, H_k]$ be a trace w.r.t. M_d and $\boldsymbol{G} = [G_0, \dots, G_k]$ be a trace w.r.t. M_t, then the strengthening of $\boldsymbol{G}$ is denoted as $\boldsymbol{G}^r = [G_0^r, \dots, G_k^r]$ such that:

$$G_0^r := Init_d \tag{12}$$

$$\begin{aligned} G_i^r(Y) :=& G_i(Y) \wedge \bar{H}_i(X) \wedge \bar{H}_i^s(X) \wedge \Big(\bigwedge\{x_t \mid H_i(Z) \rightarrow \neg(x = x_d)\}\Big) \wedge \\ & \Big(\bigwedge\{\neg x_t \mid H_i(Z) \rightarrow (x = x_d)\}\Big) \end{aligned} \tag{13}$$

The above gives us a procedure for strengthening $\boldsymbol{G}$ by using $\boldsymbol{H}$. Note that since M_t is an over-approximation of M_d, it may allow a variable $x \in X$ to be tainted, while in M_d (and therefore in $\boldsymbol{H}$), $x = x_d$. As a result, after strengthening $\boldsymbol{G}^r$ is not necessarily an inductive trace w.r.t. M_t, namely, $G_i^r \wedge Tr_t \rightarrow {G_{i+1}^r}'$ does not necessarily hold. In order to make $\boldsymbol{G}^r$ an inductive trace, M_t must be refined.

Let us assume that $G_i^r \wedge Tr_t \rightarrow {G_{i+1}^r}'$ does not hold. By that, $G_i^r \wedge Tr_t \wedge \neg {G_{i+1}^r}'$ is satisfiable. Considering the way $\boldsymbol{G}^r$ is strengthened, three exists $x \in X$ such that $G_i^r \wedge Tr_t \wedge x_t'$ is satisfiable and $G_{i+1}^r \Rightarrow \neg x_t$. The refinement step is defined by:

$$x_t' = G_i^r \; ? \; \mathit{false} \; : (\Theta(cond) \vee (cond \; ? \; \Theta(\varphi_1) \; : \; \Theta(\varphi_2)))$$

This refinement step changes the next state function of x_t such that whenever G_i holds, x_t is forced to be *false* at the next time frame.

Lemma 2. *Let $\boldsymbol{G}^r$ be a strengthened trace, and let M_t^r be the result of refinement as defined above. Then, $\boldsymbol{G}^r$ is an inductive trace w.r.t M_t^r.*

Theorem 1. *Let $\mathfrak{A}$ be a sound and complete model checking algorithm w.r.t. $FOL(\mathcal{T})$ for some $\mathcal{T}$, such that $\mathfrak{A}$ maintains an inductive trace. Assuming* IFC-CEGAR *uses $\mathfrak{A}$, then* IFC-CEGAR *is both sound and complete.*

Proof (Sketch). Soundness follows directly from the soundness of taint analysis. For completeness, assume M_d is SAFE. Due to our assumption that $\mathfrak{A}$ is sound and complete, $\mathfrak{A}$ emits a closed inductive trace $\boldsymbol{H}$. Intuitively, assuming $\boldsymbol{H}$ is of size k, then the next state function of every taint variable in M_t can be refined to be a constant *false* after a specific number of steps. Then, $\boldsymbol{H}$ can be translated to a closed inductive trace $\boldsymbol{G}$ over M_t by following the above presented formalism. Using Lemma 2 we can show that a closed inductive trace exists for the refined taint model.

5.2 IFC-BMC

In this section we introduce a different method based on Bounded Model Checking (BMC) [6] that uses lazy self-composition for solving the information flow security problem. This approach is described in Alg 2. In addition to the program P, and the specification of high-security variables H, it uses an extra parameter BND that limits

Algorithm 2: `IFC-BMC (P,H,BND)`

```
    Input: A program $P$, a set of high-security variables $H$, max unroll bound $BND$
    Output: SAFE, UNSAFE or UNKNOWN.
1   $i \leftarrow 0$
2   repeat
3       $M(i) \leftarrow$ LoopUnroll($P, i$)
4       $M_t(i) \leftarrow$ EncodeTaint($M(i)$)
5       TR of $M_s(i) \leftarrow$ LazySC($M(i), M_t(i)$)
6       Bad of $M_s(i) \leftarrow \bigvee_{y \in L} \neg(y = y')$
7       $result \leftarrow$ SolveSMT($M_s(i)$)
8       if $result = counterexample$ then
9           return UNSAFE
10      $live\_taint \leftarrow$ CheckLiveTaint($M_t(i)$)
11      if $live\_taint = false$ then
12          return SAFE
13      $i \leftarrow i + 1$
14  until $i = BND$
15  return UNKNOWN
```

Algorithm 3: `LazySC`(M_t, M)

```
    Input: A program model $M$ and the corresponding taint program model $M_t$
    Output: Transition relation of the self-composed program $Tr_s$
1   for each state update $x \leftarrow \varphi$ in transition relation of $M$ do
2       add state update $x \leftarrow \varphi$ to $Tr_s$
3       $tainted \leftarrow$ SolveSMT(query on $x_t$ in $M_t$)
4       if $tainted = false$ then
5           add state update $x' \leftarrow x$ to $Tr_s$
6       else
7           add state update $x' \leftarrow$ duplicate($\varphi$) to $Tr_s$
8   return $Tr_s$
```

the maximum number of loop unrolls performed on the program P. (Alternatively, one can fall back to an unbounded verification method after BND is reached in BMC.)

In each iteration of the algorithm (line 2), loops in the program P are unrolled (line 3) to produce a loop-free program, encoded as a transition system $M(i)$. A new transition system $M_t(i)$ is created (line 4) following the method described in section 4.1, to capture precise taint propagation in the unrolled program $M(i)$. Then lazy self-composition is applied (line 5), as shown in detail in Alg 3, based on the interplay between the taint model $M_t(i)$ and the transition system $M(i)$. In detail, for each variable x updated in $M(i)$, where the state update is denoted $x := \varphi$, we use x_t in $M_t(i)$ to encode whether x is possibly tainted. We generate an SMT query to determine if x_t is satisfiable. If it is unsatisfiable, i.e., x_t evaluates to $False$, we can conclude that high security variables cannot affect the value of x. In this case, its duplicate variable x' in the self-composed program $M_s(i)$ is set equal to x, eliminating the need to duplicate the computation that will produce x'. Otherwise if x_t is satisfiable (or unknown), we duplicate φ and update x' accordingly.

The self-composed program $M_s(i)$ created by `LazySC` (Alg 3) is then checked by a bounded model checker, where a bad state is a state where any low-security output y ($y \in L$, where L denotes the set of low-security variables) has a different value than its duplicate variable y' (line 6). (For ease of exposition, a simple definition of bad states is shown here. This can be suitably modified to account for $Obs_x(X)$ predicates described in Section 4.) A counterexample produced by the solver indicates a leak in the original program P. We also use an early termination check for BMC encoded as an SMT-based query $CheckLiveTaint$, which essentially checks whether any live variable is tainted (line 10). If none of the live variables is tainted, i.e., any initial taint from high-security inputs has been squashed, then IFC-BMC can stop unrolling the program any further. If no conclusive result is obtained, IFC-BMC will return $UNKNOWN$.

6 Implementation and Experiments

We have implemented prototypes of IFC-CEGAR and IFC-BMC for information flow checking. Both are implemented on top of SEAHORN [17], a software verification platform that encodes programs as CHC (Constrained Horn Clause) rules. It has a frontend based on LLVM [22] and backends to Z3 [14] and other solvers. Our prototype has a few limitations. First, it does not support bit-precise reasoning and does not support complex data structures such as lists. Our implementation of symbolic taint analysis is flexible in supporting any given taint policy (i.e., rules for taint generation, propagation, and removal). It uses an encoding that fully leverages SMT-based model checking techniques for precise taint analysis. We believe this module can be independently used in other applications for security verification.

6.1 Implementation Details

IFC-CEGAR *Implementation.* As discussed in Section 5.1, the IFC-CEGAR implementation uses taint analysis and self-composition synergistically and is tailored toward proving that programs are secure. Both taint analysis and self-composition are implemented as LLVM-passes that instrument the program. Our prototype implementation executes these two passes interchangeably as the problem is being solved. The IFC-CEGAR implementation uses Z3's CHC solver engine called SPACER. SPACER, and therefore our IFC-CEGAR implementation, does not handle the bitvector theory, limiting the set of programs that can be verified using this prototype. Extending the prototype to support this theory will be the subject of future work.

IFC-BMC *Implementation.* In the IFC-BMC implementation, the loop unroller, taint analysis, and lazy self-composition are implemented as passes that work on CHC, to generate SMT queries that are passed to the backend Z3 solver. Since the IFC-BMC implementation uses Z3, and not SPACER, it can handle all the programs in our evaluation, unlike the IFC-CEGAR implementation.

Input Format. The input to our tools is a C-program with annotations indicating which variables are *secret* and the locations at which leaks should be checked. In addition, variables can be marked as *untainted* at specific locations.

6.2 Evaluation Benchmarks

For experiments we used a machine running Intel Core i7-4578U with 8GB of RAM. We tested our prototypes on several micro-benchmarks[4] in addition to benchmarks inspired by real-world programs. For comparison against eager self-composition, we used the SEAHORN backend solvers on a 2-copy version of the benchmark. `fibonacci` is a micro-benchmark that computes the N-th Fibonacci number. There are no secrets in the micro-benchmark, and this is a sanity check taken from [33]. `list_4/8/16` are programs working with linked lists, the trailing number indicates the maximum number of nodes being used. Some linked list nodes contain secrets while others have public data, and the verification problem is to ensure that a particular function that operates on the linked list does not leak the secret data. `modadd_safe` is program that performs multi-word addition; `modexp_safe/unsafe` are variants of a program performing modular exponentiation; and `pwdcheck_safe/unsafe` are variants of program that compares an input string with a secret password. The verification problem in these examples is to ensure that an iterator in a loop does not leak secret information, which could allow a timing attack. Among these benchmarks, the `list_4/8/16` use structs while `modexp_safe/unsafe` involve bitvector operations, both of which are not supported by SPACER, and thus not by our IFC-CEGAR prototype.

6.3 IFC-CEGAR Results

Table 1 shows the IFC-CEGAR results on benchmark examples with varying parameter values. The columns show the time taken by eager self-composition (Eager SC) and IFC-CEGAR, and the number of refinements in IFC-CEGAR. “TO” denotes a timeout of 300 seconds.

Benchmark	**Parameter**	**Eager SC**	**IFC-CEGAR**	
		Time (s)	Time (s)	#Refinements
`pwdcheck_safe`	4	8.8	0.2	0
	8	TO	0.2	0
	16	TO	0.2	0
	32	TO	0.2	0
`pwdcheck2_safe`	$N > 8$	TO	61	1
`modadd_safe`	2048b	180	0.2	0
	4096b	TO	0.3	0

Table 1: IFC-CEGAR results (time in seconds)

We note that all examples are secure and do not leak information. Since our path-sensitive symbolic taint analysis is more precise than a type-based taint analysis, there

[4] `http://www.cs.princeton.edu/~aartig/benchmarks/ifc_bench.zip`

are few counterexamples and refinements. In particular, for our first example `pwdcheck_safe`, self-composition is not required as our path-sensitive taint analysis is able to prove that no taint propagates to the variables of interest. It is important to note that type-based taint analysis cannot prove that this example is secure. For our second example, `pwdcheck2_safe`, our path-sensitive taint analysis is not enough. Namely, it finds a counterexample, due to an implicit flow where a for-loop is conditioned on a tainted value, but there is no real leak because the loop executes a constant number of times. Our refinement-based approach can easily handle this case, where IFC-CEGAR uses self-composition to find that the counterexample is spurious. It then refines the taint analysis model, and after one refinement step, it is able to prove that `pwdcheck2_safe` is secure. While these examples are fairly small, they clearly show that IFC-CEGAR is superior to eager self-composition.

6.4 IFC-BMC Results

The experimental results for IFC-BMC are shown in Table 2, where we use some unsafe versions of benchmark examples as well. Results are shown for total time taken by eager self-composition (Eager SC) and the IFC-BMC algorithm. (As before, "TO" denotes a timeout of 300 seconds.) IFC-BMC is able to produce an answer significantly faster than eager self-composition for all examples. The last two columns show the time spent in taint checksin IFC-BMC, and the number of taint checks performed.

Benchmark	**Result**	**Eager SC**	**IFC-BMC**	**Taint checks**	**#Taint checks**
		Time (s)	Time (s)	Time (s)	
`fibonacci`	SAFE	0.55	0.1	0.07	85
`list_4`	SAFE	2.9	0.15	0.007	72
`list_8`	SAFE	3.1	0.6	0.02	144
`list_16`	SAFE	3.2	1.83	0.08	288
`modexp_safe`	SAFE	TO	0.05	0.01	342
`modexp_unsafe`	UNSAFE	TO	1.63	1.5	364
`pwdcheck_safe`	SAFE	TO	0.05	0.01	1222
`pwdcheck_unsafe`	UNSAFE	TO	1.63	1.5	809

Table 2: IFC-BMC results (time in seconds)

To study the scalability of our prototype, we tested IFC-BMC on the modular exponentiation program with different values for the maximum size of the integer array in the program. These results are shown in Table 3. Although the IFC-BMC runtime grows exponentially, it is reasonably fast – less than 2 minutes for an array of size 64.

7 Related Work

A rich body of literature has studied the verification of secure information flow in programs. Initial work dates back to Denning and Denning [15], who introduced a program analysis to ensure that confidential data does not flow to non-confidential outputs. This notion of confidentiality relates closely to: (i) non-interference introduced by Goguen and Meseguer [16], and (ii) separability introduced by Rushby [27]. Each of these study

Benchmark	Parameter	Time (s)	#Taint checks
modexp	8	0.19	180
	16	1.6	364
	24	3.11	548
	32	8.35	732
	40	11.5	916
	48	21.6	1123
	56	35.6	1284
	64	85.44	1468

Table 3: IFC-BMC results on `modexp` (time in seconds)

a notion of of secure information flow where confidential data is strictly not allowed to flow to any non-confidential output. These definitions are often too restrictive for practical programs, where secret data might sometimes be allowed to flow to some non-secret output (e.g., if the data is encrypted before output), i.e. they require declassification [29]. Our approach allows easy and fine-grained de-classification.

A large body of work has also studied the use of type systems that ensure secure information flow. Due to a lack of space, we review a few exemplars and refer the reader to Sabelfeld and Myers [28] for a detailed survey. Early work in this area dates back to Volpano et al. [34] who introduced a type system that maintains secure information information based on the work of Denning and Denning [15]. Myers introduced the JFlow programming language (later known as Jif: Java information flow) [25] which extended Java with security types. Jif has been used to build clean slate, secure implementations of complex end-to-end systems, e.g. the Civitas [9] electronic voting system. More recently, Patrigiani et al. [26] introduced the Java Jr. language which extends Java with a security type system, automatically partitions the program into secure and non-secure parts and executes the secure parts inside so-called protected module architectures.In contrast to these approaches, our work can be applied to existing security-critical code in languages like C with the addition of only a few annotations.

A different approach to verifying secure information flow is the use of dynamic taint analysis (DTA) [30,3,21,31,12,11] which instruments a program with taint variables and taint tracking code. Advantages of DTA are that it is scalable to very large applications [21], can be accelerated using hardware support [12], and tracks information flow across processes, applications and even over the network [11]. However, taint analysis necessarily involves imprecision and in practice leads to both false positives and false negatives. False positives arise because taint analysis is an overapproximation. Somewhat surprisingly, false negatives are also introduced because tracking implicit flows using taint analysis leads to a deluge of false-positives [30], thus causing practical taint tracking systems to ignore implicit flows. Our approach does not have this imprecision.

Our formulation of secure information flow is based on the self-composition construction proposed by Barthe et al. [5]. A specific type of self-composition called product programs was considered by Barthe et al. [4], which does not allow control flow divergence between the two programs. In general this might miss certain bugs as it ignores implicit flows. However, it is useful in verifying cryptographic code which typically has very structured control flow. Almeida et al. [1] used the product construction to verify that certain functions in cryptographic libraries execute in constant-time.

Terauchi and Aiken [33] generalized self-composition to consider k-safety, which uses $k - 1$ compositions of a program with itself. Note that self-composition is a 2-

safety property. An automated verifier for k-safety properties of Java programs based on Cartesian Hoare Logic was proposed by Sousa and Dillig [32]. A generalization of Cartesian Hoare Logic, called Quantitative Cartesian Hoare Logic was introduced by Chen et al. [7]; the latter can also be used to reason about the execution time of cryptographic implementations. Among these efforts, our work is mostly closely related to that of Terauchi and Aiken [33], who used a type-based analysis as a preprocessing step to self-composition. We use a similar idea, but our taint analysis is more precise due to being path-sensitive, and it is used within an iterative CEGAR loop. Our path-sensitive taint analysis leads to fewer counterexamples and thereby cheaper self-composition, and our refinement approach can easily handle examples with benign branches. In contrast to the other efforts, our work uses lazy instead of eager self-composition, and is thus more scalable, as demonstrated in our evaluation. A recent work [2] also employs trace-based refinement in security verification, but it does not use self-composition.

Our approach has some similarities to other problems related to tainting [18]. In particular, *Change-Impact Analysis* is the problem of determining what parts of a program are affected due to a change. Intuitively, it can be seen as a form of taint analysis, where the change is treated as taint. To solve this, Gyori et al. [18] propose a combination of an imprecise type-based approach with a precise semantics-preserving approach. The latter considers the program before and after the change and finds relational equivalences between the two versions. These are then used to strengthen the type-based approach. While our work has some similarities, there are crucial differences as well. First, our taint analysis is not type-based, but is path-sensitive and preserves the correctness of the defined abstraction. Second, our lazy self-composition is a form of an abstraction-refinement framework, and allows a tighter integration between the imprecise (taint) and precise (self-composition) models.

8 Conclusions and Future Work

A well-known approach for verifying secure information flow is based on the notion of self-composition. In this paper, we have introduced a new approach for this verification problem based on *lazy self-composition*. Instead of eagerly duplicating the program, lazy self-composition uses a synergistic combination of symbolic taint analysis (on a single copy program) and self-composition by duplicating relevant parts of the program, depending on the result of the taint analysis. We presented two instances of lazy self-composition: the first uses taint analysis and self-composition in a CEGAR loop; the second uses bounded model checking to dynamically query taint checks and self-composition based on the results of these dynamic checks. Our algorithms have been implemented in the SEAHORN verification platform and results show that lazy self-composition is able to verify many instances not verified by eager self-composition.

In future work, we are interested in extending lazy self-composition to support learning of quantified relational invariants. These invariants are often required when reasoning about information flow in shared data structures of unbounded size (e.g., unbounded arrays, linked lists) that contain both high- and low-security data. We are also interested in generalizing lazy self-composition beyond information-flow to handle other k-safety properties like injectivity, associativity and monotonicity.

References

1. J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium, USENIX Security.*, pages 53–70, 2016.
2. T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *PLDI*, pages 362–375, 2017.
3. G. S. Babil, O. Mehani, R. Boreli, and M. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices. In *Proceedings of Security and Cryptography*, 2013.
4. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In M. Butler and W. Schulte, editors, *FM 2011: Formal Methods*, pages 200–214, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
5. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17)*, pages 100–114, 2004.
6. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS, Proceedings*, pages 193–207, 1999.
7. J. Chen, Y. Feng, and I. Dillig. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 875–890, New York, NY, USA, 2017. ACM.
8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, CAV*, pages 154–169, 2000.
9. M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: Toward a secure voting system. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 354–368, Washington, DC, USA, 2008. IEEE Computer Society.
10. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
11. M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end Containment of Internet Worms. In *Proceedings of the Symposium on Operating Systems Principles*, 2005.
12. J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th IEEE/ACM International Symposium on Microarchitecture*, 2004.
13. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Relational Verification Through Horn Clause Transformation. In *SAS*, volume 9837, pages 147–169, 2016.
14. L. M. de Moura and N. Bjørner. 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, Proceedings*, pages 337–340, 2008.
15. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.
16. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20, 1982.
17. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 343–361, 2015.

18. A. Gyori, S. K. Lahiri, and N. Partush. Refining interprocedural change-impact analysis using equivalence relations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 318–328, 2017.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *The SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, 2002.
20. K. Hoder, N. Bjørner, and L. M. de Moura. μZ- an efficient engine for fixed points with constraints. In *Computer Aided Verification - 23rd International Conference, CAV. Proceedings*, pages 457–462, 2011.
21. M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.
22. C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO)*, pages 75–88, 2004.
23. F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. *HASP*, 13:10, 2013.
24. D. Mordvinov and G. Fedyukovich. Synchronizing Constrained Horn Clauses. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 338–355. EasyChair, 2017.
25. A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1999.
26. M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.*, 37(2):6:1–6:50, 2015.
27. J. M. Rushby. Proof of separability: A verification technique for a class of a security kernels. In *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, pages 352–367, 1982.
28. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, Sept. 2006.
29. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, Oct. 2009.
30. E. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.
31. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Information Systems Security*, 2008.
32. M. Sousa and I. Dillig. Cartesian Hoare Logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 57–69, New York, NY, USA, 2016. ACM.
33. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Static Analysis, 12th International Symposium, SAS, Proceedings*, pages 352–367, 2005.
34. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.