Noninterference via Symbolic Execution *

Dimiter Milushev, Wim Beck, Dave Clarke

IBBT-DistriNet, KU Leuven, Heverlee, Belgium

Abstract. Noninterference is a high-level security property that guarantees the absence of illicit information flow at runtime. Noninterference can be enforced statically using information flow type systems; however, these are criticized for being overly conservative and rejecting secure programs. More precision can be achieved by using program logics, but such an approach lacks its own verification tools. In this work we propose a novel, alternative approach: utilizing symbolic execution in combination with ideas from program logics in an attempt to increase the precision of analyses and automate noninterference testing. Dealing with policies incorporating declassification is also explored. The feasibility of the proposal is illustrated using a prototype tool based on the KLEE symbolic execution engine.

Keywords: Noninterference, declassification, symbolic execution

1 Introduction

Noninterference is a high-level security property, prohibiting information leaks through the executions of a program. The typical program model for expressing noninterference assumes the following: public and secret inputs are given to a program; public and secret outputs are observable as a result of the program runs. In this context, noninterference is a policy stipulating that public outputs of a program should be functionally dependent on public inputs only, and not on secret inputs. The policy has been substantially studied in the language-based security community [15] and typically relies on information flow type systems [14, 20, 21]; however, these are criticized for being conservative and rejecting many secure programs.

An alternative approach proposes the use of program logics for expressing noninterference. Such an approach was introduced by Darvas, Hähnle and Sands [8], who used dynamic logic to verify noninterference for sequential Java programs. One key observation they made is that noninterference (which is not a property and hence not directly expressible in program logics) on some program P is reducible to a property on the sequential composition $P; P'$ of the program with itself. More precisely, noninterference can be characterized as the following quadruple: $\{\bar{l} = \overline{l'}\}P; P'\{\bar{l} = \overline{l'}\}$. Here, P' is the same program as P with all

* This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`)

variables renamed, $\bar{l}$ are the low variables of P and $\overline{l'}$ are the low variables of P'. Independently, Barthe, Argenio and Rezk [4] based their characterization of noninterference in Hoare and temporal logics on similar ideas; they also coined the term *self-composition* for the construct $P; P'$. The program logics approaches provide more precision in specifications, but do not have their own verification tools; in addition, it is not clear how to reuse existing tools and techniques.

Terauchi and Aiken [19] note that self-composition is impractical. They point out that for the purpose of verification of noninterference, some nontrivial, partial-correctness condition that holds between P and P' has to be found; and finding it is impractical. They also argue that in order to be useful for practical verification, self-composition needs to take into account the structure of a self-composed program and the resulting symmetry and redundancy. They propose a type-directed transformation for a simple imperative language to deal with the problems they identify.

In this work we propose utilizing symbolic execution in combination with a form of self-composition in an attempt to automate noninterference testing. We start off with Terauchi and Aiken's transformation and accommodate additional language features, such as dealing with procedures and dynamically allocated data structures. The approach essentially interleaves two copies of a program and then uses dynamic symbolic execution to try to extract all possible paths in the program. Conditions on two disjoint program stores are generated in order to express the desired security policy via assert statements. The resultant program is analyzed by the symbolic execution engine: if the engine is able to analyze all paths, our tool can decide whether the program is secure or not. Otherwise the tool may either find a counterexample or run indefinitely.

The contributions of the work are: first, a proposal to use symbolic execution in combination with well-known program logics-based program transformations in order to specify and check a notion of plain noninterference and one incorporating declassification; second, an illustration of what is needed to transfer the ideas to a programming language having procedures and dynamic memory allocation (heap); third, a prototype tool based on the KLEE symbolic execution tool [6] illustrating the feasibility of the approach. The rest of the work is structured as follows: Section 2 provides some background. Sections 3 and 4 present the main contribution. Sections 5 and 6 give the related work and conclusion.

2 Background

2.1 Noninterference

Intuitively, noninterference stipulates that public outputs of a program should be functionally dependent on public inputs only, and not on secret inputs. Define a store $\mathtt{m}$ to be a mapping from program variables from some set *Var* to values from set $\mathcal{V}$. The notation $\mathtt{m}|_X$ is a restriction of the store to variables from domain X; $(\mathtt{m}, P)$ denotes the final store after execution of program P with initial store $\mathtt{m}$ and $(\mathtt{m}, P) = \perp$ signifies that the program diverges. Finally, $\approx_p$ signifies the

obvious pointwise extension of equality to stores. The definition of termination insensitive information flow can be formulated as follows:

Definition 1. *(Secure information flow [19]) A program P with high security variables $H = \{h_1, \ldots, h_i\}$ and low security variables $L = \{l_1, \ldots, l_j\}$ is* secure *iff for all possible stores m_1 and m_2 such that $m_1|_L \approx_p m_2|_L$, we have that*

$$((m_1, P) \neq \bot \wedge (m_2, P) \neq \bot) \implies (m_1, P)|_L \approx_p (m_2, P)|_L.$$

There is an obvious way to show that a program P is not secure by Definition 1, namely by finding two stores $\mathtt{m}_i$ and $\mathtt{m}_j$ such that $\mathtt{m}_i|_L \approx_p \mathtt{m}_j|_L$, $(\mathtt{m}_i, P) \neq \bot \wedge (\mathtt{m}_j, P) \neq \bot$, and $(\mathtt{m}_i, P)|_L \not\approx_p (\mathtt{m}_j, P)|_L$.

Program 1.1, also referred to as P_1, illustrates implicit information flow. Let $H = \{i, j\}, L = \{l\}$. Observing the value of variable l discloses whether the average of the two secret values is greater than 1000.

```
1  int average(int h1, int h2) {
2      return (h1+h2)/2; }
3  int main() {
4      int l, i, j;
5      if (average(i, j) > 1000) l = 1; else l = 0; }
```

Program 1.1. Implicit information flow

The implicit flow can be detected using Definition 1. Let $\mathtt{m}_1$ and $\mathtt{m}_2$ be such that $\mathtt{m}_1(i) = 1000, \mathtt{m}_1(j) = 900, \mathtt{m}_1(l) = 0, \mathtt{m}_2(i) = 800\ \mathtt{m}_2(j) = 1400$ and $\mathtt{m}_2(l) = 0$. We have that $\mathtt{m}_1|_L \approx_p \mathtt{m}_2|_L$, $(\mathtt{m}_1, P_1) \neq \bot \wedge (\mathtt{m}_2, P_1) \neq \bot$, but at the end of execution $(\mathtt{m}_1, P_1)(l) = 0$ and $(\mathtt{m}_2, P_2)(l) = 1$; thus $(\mathtt{m}_1, P_1)|_L \not\approx_p (\mathtt{m}_2, P_1)|_L$ implies that P_1 is insecure.

2.2 Declassification

Most useful computing systems have to release sensitive information as a part of their functionality (e.g. password checking, shopping for digital content, online games). Thus noninterference is often too strict for realistic systems; the usual solution is weakening the policy with *declassification*, a mechanism for releasing sensitive information. An important problem of declassification is to guarantee precisely *what* is being leaked and to ensure that the mechanism cannot be abused into leaking more [17].

More formally consider program P on stores $\mathtt{m}_1$ and $\mathtt{m}_2$. Let ψ be the predicate defined as $\mathtt{m}_1|_L \approx_p \mathtt{m}_2|_L$. Noninterference can be given as the following quadruple $\{\psi\}(\mathtt{m}_1, P); (\mathtt{m}_2, P)\{\psi\}$. If ψ_{decl} is a predicate on high variables specifying what is to be declassified by P, then *noninterference with declassification* is expressed as $\{\psi \wedge \psi_{decl}\}(\mathtt{m}_1, P); (\mathtt{m}_2, P)\{\psi\}$ [3].

For instance, in Program 1.1 the policy might be that it is admissible to reveal some fact about the average (i.e. whether $(average(i, j) > 1000)$ holds) but not more than that. Then, in addition to the usual noninterference condition, ψ_{decl} will be instantiated with $((average(i_1, j_1) > 1000) \Leftrightarrow (average(i_2, j_2) > 1000))$ (note that i_k is shortcut for $\mathtt{m}_k(i)$ for $k \in \{1, 2\}$).

Other dimensions of declassification are about *who* controls information release, *where* in the system does declassification occur and finally *when* can information be declassified [17]. In this work, we focus on the *what* dimension.

2.3 Symbolic execution

Symbolic execution [12] is a program analysis technique used to investigate the possible execution traces of a program. The idea is to replace program inputs with input symbols and thus instead of executing the program with concrete values, to execute it with symbolic expressions over the input symbols. When the program encounters a conditional branch statement, execution is forked because there are no concrete values to evaluate the condition: whether any or both of these branches are reachable is checked by a constraint solver. Loops can be seen as conditional statements encountered multiple times and they are lazily unrolled, possibly an infinite number of times. The conjunction of all conditions encountered on the branches of a single path is called a *path condition*.

Symbolic execution is a general approach that can be used to check or prove a range of properties of programs. Properties can be expressed using assert statements. The technique explores both the case when the assertion holds and when it does not.

Dynamic symbolic execution, also called concolic execution [18] or DART [11], is a variant of the technique interleaving concrete and symbolic execution. The idea is simple: first, gather the constraints for some path by monitoring program execution with some arbitrary, concrete inputs; then, systematically explore new execution paths by negating parts of the initial path condition.

3 Approach

3.1 Overview

The approach proposed in this paper starts with partitioning the program variables into public and secret, and respectively annotating them. Then the variables are made symbolic and a type-directed transformation adopted from the work of Terauchi and Aiken [19] is applied to the program; the transformation is a variant of the self-compositional approach. We develop certain extensions of the transformation in order to deal with aspects of procedures and dynamically allocated data structures; the latter require reasoning about the heap and a modified definition of noninterference. After the transformation is complete assertions specifying the noninterference policy have to be placed. Then the symbolic execution tool is used as a program analysis tool for noninterference. If it is able to analyze all possible paths in the transformed program, a tool based on it can decide whether the program is secure or not. Otherwise the tool may either eventually return an error, implying the program is insecure or keep running indefinitely; in the latter case the proposed approach cannot determine whether the program is secure or not.

3.2 Tool introduction

The proof-of-concept tool that we have built to validate our ideas is written in Perl and is based on KLEE: an automatic symbolic execution tool for high-coverage test generation built on top of the LLVM compiler infrastructure [6].

Our tool takes as input a specially annotated C program, performs some program transformations, adds assertions and passes the resulting program to KLEE. Based on KLEE's output, the tool can either decide whether the tested program is secure and inform the tester or keep running indefinitely if it cannot cover all paths and cannot find a counterexample in the covered part; in the latter case it is not clear whether the tested program is secure. An optional parameter may specify when to time-out and stop searching. Whenever an error is found, the *ktest-tool* tool can be used to inspect and analyze the state that caused it.

3.3 Transformation of a basic language

We start off by illustrating how to transform a program for a minimal language including variable declarations and assignments, while loops and if statements; to illustrate we work with a subset of C. Some annotations necessary to direct the transformation are identified using special comments "//#"and given next:

high The subsequent line has one or many secret variables.
assume Places an extra assumption on a variable limiting its possible values.
constant The subsequent line contains a constant.

The first step is to partition the variables and make them symbolic. It is illustrated in Program 1.2.

```
1  int l;
2  klee_make_symbolic(&l, sizeof(int), "int l");
3  //# high
4  int h;
5  klee_make_symbolic(&h, sizeof(int), "int h");
6  l = h + 5;
```

Program 1.2. Trivial noninterference example - labeled

The second step is to determine security types of expressions statically in the usual way: in essence, if an expression depends directly or indirectly on a high variable, it must be high.

The third step is to perform the necessary program transformations given in Figure 1. The rules used here are essentially Terauchi and Aiken's transformation [19] ported to a basic subset of C.

$$\frac{c \text{ atomic}}{c \to c; c'} \qquad \frac{c_1 \to c_1^\dagger \qquad c_2 \to c_2^\dagger}{c_1; c_2 \to c_1^\dagger; c_2^\dagger} \qquad \frac{b \text{ has low security type} \qquad c_1 \to c_1^\dagger \qquad c_2 \to c_2^\dagger}{\textit{if } b \textit{ then } c_1 \textit{ else } c_2 \to \textit{if } b \textit{ then } c_1^\dagger \textit{ else } c_2^\dagger}$$

$$\frac{b \text{ has low security type} \qquad c \to c^\dagger}{\textit{while } b \textit{ do } c \to \textit{while } b \textit{ do } c^\dagger} \qquad \frac{b \text{ has high security type}}{\textit{while } b \textit{ do } c \to \textit{while } b \textit{ do } c; \textit{while } b' \textit{ do } c'}$$

$$\frac{b \text{ has high security type}}{\textit{if } b \textit{ then } c_1 \textit{ else } c_2 \to \textit{if } b \textit{ then } c_1 \textit{ else } c_2; \textit{if } b' \textit{ then } c_1' \textit{ else } c_2'}$$

Fig. 1. Type-directed transformation [19]

The fourth and final stage of the transformation is to specify noninterference conditions. These are pre and post conditions derived from the program logic approach and guaranteeing that the program is secure. They assume that the low variables of the two copies are the same (and possibly some extra declassification conditions) and have to assert that the same holds at the end of the run. To illustrate the transformation approach, consider the annotated Program 1.3:

```
1        int k; int l;
2        //# high
3        int h;
4        while (k < l) {l = k; k = k+1;}
5        if (l > h) l = 1; else l = 0;
```

Program 1.3. Annotated program illustration

It is transformed into Program 1.4.

```
1        int k0; int k1;
2        klee_make_symbolic(&k0, sizeof(int), "int k0"); klee_make_symbolic(&k1, sizeof(int), "int k1");
3        int l0; int l1;
4        klee_make_symbolic(&l0, sizeof(int), "int l0"); klee_make_symbolic(&l1, sizeof(int), "int l1");
5        klee_assume(k0 == k1); klee_assume(l0 == l1);
6        //# high
7        int h0; int h1;
8        klee_make_symbolic(&h0, sizeof(int), "int h0"); klee_make_symbolic(&h1, sizeof(int), "int h1");
9        while (k0 < l0) {l0 = k0; l1 = k1; k0 = k0+1; k1 = k1+1;}
10       if (l0 > h0) l0 = 1; else l0 = 0;
11       if (l1 > h1) l1 = 1; else l1 = 0;
12       klee_assert(k0 == k1); klee_assert(l0 == l1);
```

Program 1.4. Transformed program illustration

3.4 Procedures

Whereas Terauchi and Aiken develop their transformation for a very basic language, we would like to deal with extra language features. One of these features is procedures: the rationale for transforming them is the same as for simple imperative programs. The transformation results in a new procedure with two copies of the parameters; if the original procedure has a non-void return type then two potentially different results of the same type are returned and thus have to be placed in a fresh struct.

In some cases, variant(s) of the *if* and *while* rules from Fig. 1 may have to be used. This depends on whether the procedure is called with arguments having high or low security types (or both) and is based on a respective data flow analysis. Consider Program 1.5 as an instance of a procedure to be transformed:

```
1    int checkPass(int input, int secret){
2        int access;
3        if (input == secret){access = 1; return access;}
4        else {access = 0; return access;} }
```

Program 1.5. Procedure with non-void return type

The transformed procedure should return a struct of two integers, but that means the original *return* statements have to be replaced with appropriate *goto* statements; these are used to make a transition to the second "copy" and ensure that a properly populated data structure is returned. The resulting transformation, assuming *secret* will be passed a high value (thus second version of *if* used), is:

```
1  struct intRet* checkPass2(int input0, int secret0, int input1, int secret1){
2      int access0; int access1;
3      struct intRet* intR = malloc(sizeof(struct intRet));
4      if (input0 == secret0) {access0 = 1; goto second;}
5      else {access0 = 0; goto second;}
6      second: if (input1 == secret1) {access1 = 1; goto done;}
7      else { access1 = 0; goto done;}
8      done: intR->ret0 = access0; intR->ret1 = access1;
9      return intR; }
```

Program 1.6. Transformed procedure with non-void return type

Next, we illustrate how to transform the *int result* = *checkPass*(*guess*, *pass*) procedure call (assuming it is the whole program). Note that r is a "return" struct with two integer fields:

```
1  int result0; int result1; klee_assume(result0 == result1);
2  struct intRet* r = checkPass2(guess0, pass0, guess1, pass1);
3  result0 = r->ret0; result1 = r->ret1;
4  klee_assert(result0 == result1);
```

3.5 Dynamically allocated data structures and noninterference

It has already been suggested that different parts of a struct can be high or low. In order to model this and use symbolic execution to check programs allocating memory on the heap, we need to model the heap and change the noninterference definition respectively.

Let F be a set of fields, $\mathcal{L}$ a set of locations and $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{null\}$ be a set of values. A heap $\mathtt{h}$ will be modeled, following prior work [5], as a partial function $\mathtt{h} : \mathcal{L} \rightharpoonup S$; here $S = F \rightharpoonup \mathcal{V}$ is another partial function that models structs; the set of all heaps is *Heap*. Now $(\mathtt{m}, \mathtt{h}, P)$ denotes the final state after executing program P with store $\mathtt{m}$ and heap $\mathtt{h}$. We write $(\mathtt{m}, \mathtt{h}, P) = (\mathtt{m}_f, \mathtt{h}_f)$ to mean that the state evaluates to store $\mathtt{m}_f$ and heap $\mathtt{h}_f$. Let β be a partial bijection on memory locations, used to model the low observer's uncertainty [2]. Let $v, v' \in \mathcal{V}$, L and H be the low and high elements respectively of a typical security lattice. *Value indistinguishability* [5] can be defined as follows:

$$null \sim_{\beta,L} null \qquad v \sim_{\beta,H} v' \qquad \frac{v \in \mathbb{Z}}{v \sim_{\beta,L} v} \qquad \frac{l, l' \in \mathcal{L} \quad \beta(l) = l'}{l \sim_{\beta,L} l'}.$$

Intuitively, two heaps $\mathtt{h}_1$ and $\mathtt{h}_2$ are indistinguishable if there is a bijection that relates each struct s_1 in heap $\mathtt{h}_1$ to its counterpart s_2 in heap $\mathtt{h}_2$; the structs have the same fields (because they are of the same type) and moreover the values of corresponding fields are indistinguishable. This is formally defined as follows: two heaps $\mathtt{h}_1$, $\mathtt{h}_2$ are indistinguishable w.r.t. bijection β denoted $\mathtt{h}_1 \sim_\beta \mathtt{h}_2$ whenever: (1) $dom(\beta) \subseteq dom(\mathtt{h}_1)$ and $rng(\beta) \subseteq dom(\mathtt{h}_2)$; (2) for all $s \in dom(\beta)$ we have that $dom(\mathtt{h}_1(s)) = dom(\mathtt{h}_2(\beta(s)))$ (for every struct in $\mathtt{h}_1$ its corresponding by β struct in $\mathtt{h}_2$ has the same fields) and (3) for all fields $f \in dom(\mathtt{h}_1(s))$ with security level L we have that $\mathtt{h}_1(s)(f) \sim_{\beta,L} \mathtt{h}_2(\beta(s))(f)$, i.e. all field values of β-corresponding structs are L-indistinguishable. Similarly all fields with security level H in corresponding structs have field values that are H-indistinguishable.

Definition 2. *(Secure information flow [5]) A program P is* secure *iff for all possible stores $m, m' \in Var \to \mathcal{V}$ and heaps $h, h', h_f, h'_f \in Heap$, and partial bijection β such that $(m, h, P) \neq \perp$ and $(m', h', P) \neq \perp$, and $(m, h, P) = (m_f, h_f)$ and $(m', h', P) = (m'_f, h'_f)$, and $m \sim_\beta m'$ and $h \sim_\beta h'$ imply $m_f \sim_{\beta'} m'_f$ and $h_f \sim_{\beta', L} h'_f$ for partial bijection $\beta' \supseteq \beta$.*

The condition $\beta' \supseteq \beta$ actually models the fact that new data structures may be dynamically created at runtime and thus the bijection may become larger. An illustration of the use of the new definition follows. The main function of a simplistic e-banking program is given in Program 1.7.

```
1  int main() {
2      struct bank* bank = createBank(); struct account* account = createAccount(bank);
3      //# high
4      int amount = 100;
5      addToBalance(account, amount); }
```

Program 1.7. Banking program - main

Each procedure call on line 2 declares and creates a struct. Each transformed procedure creates a pair of structs "packed" in another struct (see Section 3.4). The public fields of the struct are assumed equal. The result of the transformation is Program 1.8.

```
1  int main() {
2      struct bankRet* bankr = createBank2();
3      klee_assume(bankr->bank0->count == bankr->bank1->count);
4
5      struct accountRet* accr = createAccount2(bankr->bank0, bankr->bank1);
6      klee_assume(accr->account0->wealthy == accr->account1->wealthy);
7      klee_assume(accr->account0->id == accr->account1->id);
8
9      int amount0; int amount1;
10     klee_make_symbolic(&amount0, sizeof(int), "int amount0");
11     klee_make_symbolic(&amount1, sizeof(int), "int amount1");
12
13     addToBalance2(accr->account0, amount0, accr->account1, amount1);
14
15     klee_assert(bankr->bank0->count == bankr->bank1->count);
16     klee_assert(accr->account0->wealthy == accr->account1->wealthy);
17     klee_assert(accr->account0->id == accr->account1->id); }
```

Program 1.8. Banking program - main transformation

In summary, whenever a new struct is allocated on the heap, the respective transformation allocates two structs and makes the appropriate assumptions about the low fields of the struct. At the end of execution, the respective assertions about the low structs have to hold.

4 Experimental Results

4.1 Implicit flow, explicit flow or no flow

Consider Program 1.9: it would be rejected as insecure by a typical information flow type system.

```
1      int l;
2      //# high
3      int h, j;
4      if ( (j + h) > 999 ) l = -1; l = h; l = l - h;
```

Program 1.9. Secure program rejected by flow-sensitive type systems

If we were to consider the program until and including the *if* statement there would be an implicit flow; the program until and including the following statement ($l = h$) would have both implicit and explicit flows. But Program 1.9 is secure: closer inspection shows that the leaks "neutralize" each other. The results are confirmed by our tool: it terminates and no counterexamples are generated.

4.2 e-Banking example

The e-banking program of Section 3.5 is presented next. The interesting, security-related code is in procedure *addToBalance2*:

```
1  if (amount >= 10000) account->wealthy = true; else account->wealthy = false;
```

Program 1.10. Security-related part of e-Banking example

Whenever the balance gets higher than 10000, flag *wealthy* is set. The field *wealthy* of the struct *account* is a public variable and leaks information about the balance. The latter is confirmed by our tool, producing the following output:

```
ebank.c:118:ASSERTION FAIL: accr->account0->wealthy == accr->account1->wealthy
```

4.3 Average example

Recall that Program 1.1 computes the average of two high variables. Assumptions on the values of variables, such as *//#assume (*$i > 0$ & $j > 0$*)*, can be specified. As already discussed, the program is not secure and the tool terminates with another assertion fail error of the condition ($l0 == l1$). It should be noted that our tool is precise in the sense that the errors are reproducible. The values that broke the assertions can be inspected using KLEE's *ktest-tool* in order to analyze the problem. The values generated by the *ktest-tool* are: $l0 = 0$, $l1 = 0$, $i0 = 506$, $i1 = 1609415267$, $j0 = 507$, $j1 = 485081005$.

4.4 Password example

Next consider the simple password check in Program 1.11.

```
1    int access, input;
2    //# high
3    int pass;
4    //# declassify (input == pass)
5    if (input == pass) access = 1; else access = 0;
```

Program 1.11. Password check — insecure

Password checking programs leak information as a part of their functionality. This can be seen if we consider the program without line 4; even when a given guess is wrong, guessing reveals that the password is or is not equal to the guess. This trivial leak is detected as another assertion fail error on condition ($access0 == access1$). As a result of the declassify statement though, the extra

condition $((input0 == pass0) == (input1 == pass1))$ is added to the the assumptions. In this case, our tool verifies that Program 4.4 is secure.

Our tool handles various examples of explicit and implicit information flow as well as a notion of information release, typically detectable by information flow type systems. Because the transformations are based on semantic methods, the approach is more precise than information flow type systems resulting in the lack of false positives. We envision that the approach could be used together with a type system: first, the type system checks the program; then, if the program is rejected, our approach could be used to further investigate the error and try to find out if the type system was possibly too strict and the program is secure. On the other hand, the approach suffers from traditional weaknesses of symbolic execution, such as problems with scalability for large numbers of paths, dependence on the power of the constraint solver and difficult interaction with the environment. Moreover, the approach will benefit from further development of test input generation methods for programs with pointers.

5 Related work

There has been a substantial amount of work on verifying noninterference from the language-based security community (see a survey [15]). Solutions traditionally relied on information flow type systems, a syntactic approach which tends to be too conservative in practice. On the other hand, many attempts to address noninterference have a semantic flavor [9, 4]; such approaches are attractive because they suggest methods to transform the problem so as to benefit from state-of-the-art verification techniques and tools. These gave rise to further work on program-logics based characterizations of noninterference [8, 4]: both these rely on the idea of reducing noninterference of a program to a property of the sequential composition of the program with itself. Reasoning about such constructs is facilitated by Terauchi and Aiken's type-directed transformation [19], which takes advantage of the structure of a self-composed program and the resulting symmetry and redundancy.

In the most relevant related work Backes et al. [1] use techniques similar to ours to compute all information leaks in a program and to quantify the leaks using information-theoretic means. Flows in their work are characterized by an equivalence relation on secrets and can be expressed as a logical assertion on program variables; this is similar to our approach. They start with a relation expressing noninterference and gradually refine it, when counterexamples are found. Their quantitative analysis is based on computing the number and sizes of equivalence classes. Although our techniques are similar, our goals are different. First, we address a different problem: deciding whether a program conforms with a base-line information flow policy, possibly augmented with a notion of information release; second, we use symbolic execution, whereas their approach uses off-the-shelf model checkers; finally, we explore qualitative policies.

Declassification is also a well-studied topic (see [17] for an overview). The idea to use equivalence relations to characterize partial information flow was

originally proposed by Cohen [7] and further developed in the literature [22, 10]. A number of related articles explore the use of equivalence relations to characterize information release using flow-sensitive type systems [13, 16]

6 Conclusion

We have presented a semi-automated approach to testing noninterference: the only non-automated phase is the initial identifying of the secrets in a candidate program and adding appropriate annotations. Dynamic symbolic execution is then used to extract all possible paths and try to break the assertions. Whenever the symbolic execution tool is able to analyze all possible paths, the proposed approach can decide whether the program is secure or not. Even if not all paths can be covered, the approach in general is useful for testing; a major advantage is precision: any assert violation indicates a concrete, reproducible security bug. On the other hand, the proposed approach suffers from the aforementioned limitations of symbolic execution.

To illustrate the usefulness of the proposed approach, we have built a prototype tool based on the symbolic execution tool KLEE. Our tool takes as input an annotated C program, performs the necessary program transformations and passes the resulting program to KLEE; then, based on KLEE's output the tool informs whether the program is secure, not secure or if it cannot decide. We have verified a number of small programs exhibiting known patterns of insecurity.

We are aware of the limitations of the approach. At present, our definitions explore termination-insensitive noninterference only; additionally, only initial and final states are observable in the model; it should be noted that these limitations are not exclusive to this work but shared by a vast body of research. In addition, the scalability of the approach for large programs still needs to be explored. Finally, we have not presented formal soundness proof. Future work will address these limitations.

References

1. Michael Backes, Boris Kopf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 141–153, Washington, DC, USA, 2009. IEEE Computer Society.
2. Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005.
3. Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Towards a logical account of declassification. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, PLAS '07, pages 61–66, New York, NY, USA, 2007. ACM.
4. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, pages 100–114, Washington, DC, USA, 2004. IEEE Computer Society.

5. Gilles Barthe and Tamara Rezk. Non-interference for a JVM-like language. TLDI '05, pages 103–112, New York, NY, USA, 2005. ACM.
6. Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.
7. E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
8. Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *SPC*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer, 2005.
9. Riccardo Focardi and Roberto Gorrieri. A taxonomy of security properties for process algebras. *Journal of Computer Security*, 3(1):5–34, 1995.
10. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 186–197. ACM-Press, NY, 2004.
11. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.
12. James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385–394, July 1976.
13. Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.
14. François Pottier and Vincent Simonet. Information flow inference for ML. *SIGPLAN Not.*, 37:319–330, January 2002.
15. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
16. Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *In Proc. International Symp. on Software Security (ISSS03), volume 3233 of LNCS*, pages 174–191. Springer-Verlag, 2004.
17. Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 255–269, Washington, DC, USA, 2005. IEEE Computer Society.
18. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Software Engineering Notes*, 30:263–272, September 2005.
19. Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *Static Analysis Symposium/Workshop on Static Analysis*, pages 352–367, 2005.
20. Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. TAPSOFT '97, pages 607–621, London, UK, 1997. Springer-Verlag.
21. S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 29 – 43, July 2003.
22. Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, CSFW '01, pages 15–23, Washington, DC, USA, 2001. IEEE Computer Society.