Self-composition by Symbolic Execution

Quoc-Sang Phan

Queen Mary University of London
qsp30@eecs.qmul.ac.uk

Abstract

Self-composition is a logical formulation of *non-interference*, a high-level security property that guarantees the absence of illicit information leakages through executing programs. In order to capture program executions, self-composition has been expressed in Hoare or modal logic, and has been proved (or refuted) by using theorem provers. These approaches require considerable user interaction, and verification expertise. This paper presents an automated technique to prove self-composition. We reformulate the idea of self-composition into comparing pairs of symbolic paths of the same program; the symbolic paths are given by Symbolic Execution. The result of our analysis is a logical formula expressing self-composition in first-order theories, which can be solved by off-the-shelf Satisfiability Modulo Theories solvers.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Information Flow, Symbolic Execution, Satisfiability Modulo Theories

Digital Object Identifier 10.4230/OASIcs.ICCSW.2013.95

1 Secure information flow: from *non-interference* to *self-composition*

Information flow in an information theoretical context is the transfer of information from a variable H to a variable O in a given process. The simplest case of information flow is *explicit flow* (or direct flow) where the whole or partial value of H is copied directly to O, for example:

```
O = H + 3;
```

There are more subtle cases, which are categorized as *implicit flow* (or indirect flow). Consider, for example, the program below, which simulates a common password checking procedure:

Listing 1 a password checking program

```
if (H == L)
    O = true;
else
    O = false;
```

H is the password, i.e. the confidential data; L is the public input provided by the user; O is the observable output, $O = true$ means the password is accepted. Although H is not directly copied to O, there is still information flow leaked $H \rightarrow O$. This information is "small", but one can reveal all information about H if he is allowed to make enough attempts.

Obviously, information flow from confidential data to observable output is not desirable, which is the motivation of research in *secure information flow*. Dating back to the pioneering work of the Dennings in the 1970s [4], secure information flow analysis has been an active research topic for the last four decades.

Non-interference. A popular security policy that guarantees the absence of information flow leaks is non-interference [2, 5]. It is stated as follows: suppose a program P takes secret input H, public input L and produces public output O. Then P satisfies the non-interference property iff the value of O does not depend on H.

There has been a large body of work that has used type systems for validating non-interference, following the idea of Volpano et al. [10]. Type systems are fast and the analysis is *safe*, which means a program is classified as "*secure*", then it is actually secure, there are no false negatives. However, they also return too many false positives, which means secure programs can be classified as "*insecure*". For example, consider again the two examples with a small modification to make them satisfy non-interference:

Listing 2 A trivial secure program

```
O = H - H + 3;
```

Listing 3 An "always-reject" password checking program

```
if (H == L)
    O = false;
else
    O = false;
```

Typing rules would always classify programs like the above as insecure. Another tricky case is of programs that leak information in the intermediate states, but sanitize information at the end, for example:

```
O = H + 3;
O = 3;
```

Given that the attacker can only observe the final value of the output O, the program is secure. However, it would be classified as insecure by type systems.

Self-composition. Another prominent approach for secure information flow is to use theorem proving, in which non-interference is logically formulated as *self-composition* [3, 1], as non-interference itself is not a logical property.

We assume a similar setting as in the case of non-interference: given a program P that takes secret input H, public input L and producing public output O, we denote by P_1 the same program as P, with all variables renamed: H as H_1, L as L_1 and O as O_1. For example, consider again the password checking program P in Listing 1, the composition of P and its copy P_1 is as follows:

```
if (H == L)
    O = true;
else
    O = false;
/* copy of the same program with all variables renamed */
if (H1 == L1)
    O1 = true;
else
    O1 = false;
```

Self-composition is expressed in Hoare-style framework as [1]:

$$\{L = L_1\}P; P_1\{O = O_1\} \tag{1}$$

The Hoare triple states that if the precondition $L = L_1$ holds, then after the execution of $P; P_1$, the postcondition $O = O_1$ also holds. Recall that non-interference requires the output O not to depend on the secret input H, which means that for any pair of possible executions of P that only differ in H, they have to agree on the public output O. In self-composition, the purpose of having the copy P_1 with all variables renamed is to have *another* P to compare with P, so self-composition is logical formulation of non-interference.

For the example above, by choosing $H = L \wedge H_1 \neq L_1$, it is easy to find a counterexample for the Hoare triple in (1), such that $L = L_1$ holds and $O = O_1$ does not hold. Therefore, the password checking program violates self-composition, and hence there is information leaked $H \rightarrow O$.

Compared to type system approach, the theorem proving approach is much more precise, returning no false positives. However, it is impractical in reality, as elegantly put in [9] by Terauchi and Aiken:

"When we actually applied the self-composition approach, we found that not only are the existing automatic safety analysis tools not powerful enough to verify many realistic problem instances efficiently (or at all), but also that there are strong reasons to believe that it is unlikely to expect any future advance."

Terauchi and Aiken also pointed out that the limitations of self-composition come from the symmetry and redundancy of the self-composed program, which lead to some partial-correctness conditions that hold between P and P_1. To find these conditions is crucial for the effectiveness of the analysis, however, finding them is in general impractical.

Moreover, to prove (or refute) self-composition with theorem provers requires considerable user interaction and verification expertise [3].

Contribution. This paper presents an automated technique for non-interference based on self-composition. The self-composition approach can be divided into two steps: first, to compose the program with a copy of itself; second, to perform analysis on the self-composed program. Our approach is to delay self-composing to the second step: first, we perform analysis on the original program with Symbolic Execution; second, we self-compose the result of the analysis to get the formula of self-composition. The idea of self-composition is to have a copy P_1 of P to compare with itself. We expand this idea into comparing all pairs of executions ρ of P and ρ_1 of P_1. Since it is impossible to enumerate all possible executions, we use Symbolic Execution to synthesize the symbolic paths that represents a set of concrete executions, and perform comparison on these symbolic paths, which we formulate as *path-equivalence*.

The delay of self-composing after performing the analysis is the main novelty of our approach. In this way, we could avoid the symmetry and redundancy of the self-composed program. Moreover, the symbolic paths synthesized by Symbolic Execution are presented by first-order theories, just as the generated formula of self-composition. The validity of this formula can be automatically and efficiently checked by powerful Satisfiability Modulo Theories (SMT) solvers.

2 Preliminaries

A deterministic program is modelled as a transition system:

$$P = (\Sigma, I, F, T)$$

where Σ is the set of program states; $I \subseteq \Sigma$ is the set of initial states; $F \subseteq \Sigma$ is the set of final states; and $T \subseteq \Sigma \times \Sigma$ is the transition function. Under this setting, a trace of (concrete) execution of program P is represented by a sequence of states:

$$\rho = \sigma_0\sigma_1..\sigma_n$$

such that $\sigma_0 \in I, \sigma_n \in F$ and $\langle \sigma_i, \sigma_{i+1}\rangle \in T$ for all $i \in \{0,..,n-1\}$. We define two functions *init* and *fin* to get the initial state and final state of ρ:

$$init(\rho) = \sigma_0 \text{ and } fin(\rho) = \sigma_n$$

The semantics of P is then defined as the set $\mathcal{R}$ of all possible traces.

We assume that each initial state $\sigma \in I$ is a pair $\langle H, L\rangle$, i.e. $I = I_H \times I_L$, in which H is the confidential component to be protected and L is the public component that may be controlled by an attacker.

Symbolic Execution. Symbolic Execution (SE), first introduced by King in the 1970s [6], is a technique widely used in verification and testing. The key idea is the following. Instead of taking inputs to be concrete values, SE takes inputs to be symbols, e.g. α, β, which represent sets of concrete input values; the program is then executed just like in normal execution. In the setting of SE, the program P is modelled as a transition system:

$$P = (\Sigma^s, I^s, F^s, T^s)$$

where Σ^s is the set of symbolic states; each $\sigma^s \in \Sigma^s$ represents a set of concrete states $\sigma \in \Sigma$. $I^s \subseteq \Sigma^s$ is the set of initial symbolic states; $F^s \subseteq \Sigma^s$ is the set of final symbolic states; and $T^s \subseteq \Sigma^s \times \Sigma^s$ is the transition function. A symbolic path (symbolic trace) of the program P is represented by a sequence of symbolic states:

$$\rho^s = \sigma_0^s\sigma_1^s..\sigma_n^s$$

such that $\sigma_0^s \in I^s, \sigma_n^s \in F^s$ and $\langle \sigma_i^s, \sigma_{i+1}^s\rangle \in T^s$ for all $i \in \{0,\ldots,n-1\}$. The symbolic semantics of P is then defined as the set of all symbolic paths $\mathcal{R}^s$, which is also called as the *symbolic execution tree.* Likewise, each $\rho^s \in \mathcal{R}^s$ represents a set of traces $\rho \in \mathcal{R}$.

We denote by $X|_y$ the value of the variable X at the state y. After symbolically executing the program P with initial input symbols $H = \alpha, L = \beta$, for each $\sigma_i^s \in F^s$, i.e. each leaf of the symbolic execution tree, we have a symbolic formula for the value of the output O in the symbolic environment:

$$O|_{\sigma_i^s} = f_i(\alpha, \beta)$$

Another product of SE is the path condition $pc_i \equiv c_i(\alpha,\beta)$ for σ_i^s to be reachable. Each pc_i corresponds to a symbolic path ρ_i^s. The following theorem was also proved by King [6]:

▶ **Theorem 1.**

$$\forall i,j \in [1,n] \wedge i \neq j . pc_i \wedge pc_j = \bot$$

We define the function *path* such that:

$$path(\rho_i^s) = pc_i$$

The output O can be considered as a result of the following function:

$$O = \left\{ \begin{array}{ll} f_1(\alpha,\beta) & \text{if } c_1(\alpha,\beta) \\ f_2(\alpha,\beta) & \text{if } c_2(\alpha,\beta) \\ \ldots & \ldots \\ f_n(\alpha,\beta) & \text{if } c_n(\alpha,\beta) \end{array} \right\} \tag{2}$$

Or the following always holds:

▶ **Corollary 2.**

$$\forall i \in [1, n].c_i(\alpha, \beta) \rightarrow O = f_i(\alpha, \beta)$$

f_i and c_i are in general combination of first-order theories, e.g. *linear arithmetic*, *bit vector* and so on. SE tools make use of off-the-shelf SMT solvers to check the satisfiability of c_i, and eliminate unreachable paths (which may appear in the control flow graph).

3 Self-composition by Symbolic Execution

To avoid the limitation of the theorem proving approach, we need to reformulate the self-composition formula into a simpler logic which does not contain the program P. This is made possible by using the trace semantics of programs.

3.1 Self-composition as path-equivalence

Given a program P that takes secret input H, public input L and producing public output O; P_1 is the same program as P, with all variables renamed: H as H_1, L as L_1 and O as O_1. The trace semantics of P and P_1 are $\mathcal{R}$ and $\mathcal{R}_1$ respectively.

▶ **Definition 3** (trace-equivalence)**.** The program P satisfies non-interference if:

$$\forall \rho \in \mathcal{R}, \rho_1 \in \mathcal{R}_1.L|_{init(\rho)} = L_1|_{init(\rho_1)} \rightarrow O|_{fin(\rho)} = O_1|_{fin(\rho_1)} \tag{3}$$

It is stated similarly to the Hoare triple in (1): for all possible pairs of traces ρ of P, and ρ_1 of P_1: if $L = L_1$ at the initial states, then $O = O_1$ at the final states. At this point, we have a formulation of self-composition that does not involve the programs P and P_1.

However, even with simple programs, it is impossible to compute all the traces. Our solution is to use trace-equivalence with SE. Recall that each symbolic path represents a set of traces, and it is possible to build a complete symbolic execution tree (here we only consider bounded programs). Following Corollary 2, trace-equivalence in the context of SE is redefined as follows:

▶ **Definition 4** (path-equivalence)**.** The program P satisfies non-interference if:

$$\forall \rho^s \in \mathcal{R}^s, \rho_1^s \in \mathcal{R}_1^s.(L|_{init(\rho^s)} = L_1|_{init(\rho_1^s)}) \wedge path(\rho^s) \wedge path(\rho_1^s) \rightarrow (O|_{fin(\rho^s)} = O_1|_{fin(\rho_1^s)}) \tag{4}$$

In this way, we have an SMT formula, i.e. a combination of first-order theories. This is the key novelty of our approach, since the formulation of self-composition in first-order theories enables us to solve it efficiently using off-the-shelf SMT solvers.

3.2 Path-equivalence generation

Suppose P is symbolically executed with $H = \alpha, L = \beta$. To simplify the formula, we choose the input symbols for P_1 as $H_1 = \alpha_1, L_1 = \beta$ so that $L|_{init(\rho^s)} = L_1|_{init(\rho_1^s)}$ is automatically satisfied. That means:

$$(H|_{init(\rho^s)} = \alpha) \wedge (L|_{init(\rho^s)} = \beta) \wedge (H_1|_{init(\rho_1^s)} = \alpha_1) \wedge (L_1|_{init(\rho_1^s)} = \beta)$$

Given the result of SE is a function of the output O as in (2), the path-equivalence in (4) can be rewritten as:

$$PE \equiv DF \wedge IF$$

where:

$$DF \equiv \bigwedge_{i=1}^{n} c_i(\alpha,\beta) \wedge c_i(\alpha_1,\beta) \rightarrow (f_i(\alpha,\beta) = f_i(\alpha_1,\beta)) \tag{5}$$

$$IF \equiv \bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} c_i(\alpha,\beta) \wedge c_j(\alpha_1,\beta) \rightarrow (f_i(\alpha,\beta) = f_j(\alpha_1,\beta)) \tag{6}$$

DF checks the path-equivalence when both P and P_1 follow the same symbolic path, and thus it guarantees the absence of direct flows. On the other hand, IF checks the path-equivalence when P and P_1 follow different symbolic paths, and it guarantees the absence of implicit flows.

4 Case Studies

We illustrate the approach with some toy examples. Here we assume the same setting as above: a program P with confidential input H, public input L, and output O. SE executes P with input symbols $H = \alpha$ and $L = \beta$.

4.1 Implicit flow

Consider the password checking program in Listing 1. By SE, we have:

$$O = \left\{ \begin{array}{ll} true & \text{if } \alpha = \beta \\ false & \text{if } \alpha \neq \beta \end{array} \right\}$$

DF and DF are generated as follows:

$$DF \equiv (\alpha = \beta \wedge \alpha_1 = \beta \rightarrow true = true) \wedge (\alpha \neq \beta \wedge \alpha_1 \neq \beta \rightarrow false = false)$$
$$IF \equiv \alpha = \beta \wedge \alpha_1 \neq \beta \rightarrow true = false$$

It is trivial to prove that DF is valid and IF is invalid, and thus the program violates non-interference and leaks information via implicit flows.

4.2 No flow

Consider the modified version of the password checking procedure in Listing 3. By SE, we have:

$$O = \left\{ \begin{array}{ll} false & \text{if } \alpha = \beta \\ false & \text{if } \alpha \neq \beta \end{array} \right\}$$

DF and IF are generated as follows:

$$DF \equiv (\alpha = \beta \wedge \alpha_1 = \beta \rightarrow false = false) \wedge (\alpha \neq \beta \wedge \alpha_1 \neq \beta \rightarrow false = false)$$
$$IF \equiv \alpha = \beta \wedge \alpha_1 \neq \beta \rightarrow false = false$$

It is trivial to prove that both DF and IF are valid, and thus the program satisfies non-interference. Note that this is the case that type systems, taint analysis would decide as violating non-interference.

No confidential data involved. Consider again the password checking program, with a small modification to exclude the confidential data in its computation, i.e. to make it secure.

Listing 4 A program without confidential data

```
if (L == 3)
    O = true;
else
    O = false;
```

Similarly we have:

$$O = \left\{ \begin{array}{ll} \mathit{true} & \text{if } \beta = 3 \\ \mathit{false} & \text{if } \neg(\beta = 3) \end{array} \right\}$$

DF and *IF* are derived as:

$$\begin{aligned} DF &\equiv (\beta = 3 \land \beta = 3 \rightarrow \mathit{true} = \mathit{true}) \land (\neg(\beta = 3) \land \neg(\beta = 3) \rightarrow \mathit{false} = \mathit{false}) \\ IF &\equiv \beta = 3 \land \neg(\beta = 3) \rightarrow \mathit{true} = \mathit{false} \end{aligned}$$

Both *DF* and *IF* are valid, which confirms the intuition that the program is secure.

4.3 Both implicit and explicit flows

Consider the following data sanitization program:

```
if (H < 16)
    O = H + L;
else
    O = L;
```

The summaries and path conditions returned by SE are as follows:

$$O = \left\{ \begin{array}{ll} \alpha + \beta & \text{if } \alpha < 16 \\ \beta & \text{if } \neg(\alpha < 16) \end{array} \right\}$$

DF and *DF* are generated similarly:

$$\begin{aligned} DF &\equiv (\alpha < 16 \land \alpha_1 < 16 \rightarrow \alpha + \beta = \alpha_1 + \beta) \land (\neg(\alpha < 16) \land \neg(\alpha_1 < 16) \rightarrow \beta = \beta) \\ IF &\equiv \alpha < 16 \land \neg(\alpha_1 < 16) \rightarrow \alpha + \beta = \beta \end{aligned}$$

It is easy to find counterexamples to make *DF* and *IF* invalid, for example: $(\alpha = 1; \alpha_1 = 2)$ for *DF* and $(\alpha = 1; \alpha_1 = 17)$ for *IF*. So the program leaks via both implicit and explicit flows.

5 Related Work

Self-composition was first introduced by Darvas et al. [3] who expressed it in dynamic logic and proved information flow properties for Java CARD programs. Their approach is not automated, requiring users to provide loop invariants, induction hypotheses and so on. Barthe et al. [1] then coined the term “self-composition” and investigated its theoretical aspects, extending the problem to non-deterministic and termination-sensitive cases.

Terauchi and Aiken [9] found that self-composition was problematic, since the self-composed programs contains symmetry and redundancy. They proposed a type-directed transformation for a simple imperative language to deal with the problem. Milushev et al.

[7] implemented this type-directed transformation and used *Dynamic Symbolic Execution* (also known as *concolic testing*) as a program analysis tool for non-interference.

To our knowledge, our technique is unique in that it only performs analysis on the original program, rather than the self-composed program, the idea of self-composition is shown in the way we rename the symbolic formula, not in the analysis stage.

In our previous work [8], we proposed Symbolic Quantitative Information Flow (SQIF), an approach that uses Symbolic Execution to "measure" information flow leaks, i.e. quantitative information flow.

6 Conclusion

We have presented an automated method for secure information flow analysis. We build our work on the classical self-composition approach. However, instead of performing the analysis on the self-composed program, we use SE on the original program. This is enabled by reformulating self-composition into path-equivalence, a property for symbolic paths returned by SE.

Acknowledgements. We thank Nikos Tzevelekos and the anonymous reviewers for constructive comments.

References

1 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*, CSFW '04, pages 100–, Washington, DC, USA, 2004. IEEE Computer Society.

2 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.

3 Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In *Proceedings of the Second international conference on Security in Pervasive Computing*, SPC'05, pages 193–209, Berlin, Heidelberg, 2005. Springer-Verlag.

4 Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

5 Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

6 James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

7 Dimiter Milushev, Wim Beck, and Dave Clarke. Noninterference via symbolic execution. In *Proceedings of the 14th joint IFIP WG 6.1 international conference and Proceedings of the 32nd IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 152–168, Berlin, Heidelberg, 2012. Springer-Verlag.

8 Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Păsăreanu. Symbolic quantitative information flow. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, November 2012.

9 Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *Proceedings of the 12th international conference on Static Analysis*, SAS'05, pages 352–367, Berlin, Heidelberg, 2005. Springer-Verlag.

10 Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.