Secure Information Flow as a Safety Problem *

Tachio Terauchi[1] and Alex Aiken[2]

[1] EECS Department, University of California, Berkeley
[2] Computer Science Department, Stanford University

Abstract. The termination insensitive secure information flow problem can be reduced to solving a safety problem via a simple program transformation. Barthe, D'Argenio, and Rezk coined the term "self-composition" to describe this reduction. This paper generalizes the self-compositional approach with a form of information downgrading recently proposed by Li and Zdancewic. We also identify a problem with applying the self-compositional approach in practice, and we present a solution to this problem that makes use of more traditional type-based approaches. The result is a framework that combines the best of both worlds, i.e., better than traditional type-based approaches and better than the self-compositional approach.

1 Introduction

A termination insensitive secure information flow problem can be defined as follows:

Definition 1 (Secure Information Flow). *Given a program P whose variables $H = \{h_1, h_2, \ldots, h_n\}$ are high security variables and $L = \{l_1, \ldots, l_n\}$ are low-security variables, P is said to be secure if and only if the values of L at the point P terminates are independent of the initial values of H.*

In this paper, we only deal with the case where programs are deterministic. The secure information flow problem is a type of non-interference problem. In practice, it expresses the problem of whether some selected information in a program or a fragment of a program (i.e., the information stored in the high-security variables) does not leak to an adversary (i.e., the low-security variables). Secure information flow has applications in software security. There is an excellent survey by Sabelfeld and Myers on issues ranging from applications to analysis techniques [1]. We note that the definition above can be extended to multi-label cases (i.e., beyond just "high" and "low") by posing the problem multiple times with different choices of high-security variables and low-security variables.

An equivalent way to state the termination insensitive secure information flow problem is:

* This research was supported in part by NASA Grant No. NNA04CI57A; NSF Grant Nos. CCR-0234689, CCR-0085949, and CCR-0326577. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Definition 2 (Secure Information Flow - Alternative Definition). *Given a program P whose variables $H = \{h_1, \ldots, h_n\}$ are high security variables and $L = \{l_1, \ldots, l_n\}$ are low-security variables, P is said to be secure if and only if for any stores M_1 and M_2 such that $M_1|_{H^c} = M_2|_{H^c}$,*

$$(\langle M_1, P\rangle \neq \bot \wedge \langle M_2, P\rangle \neq \bot) \Rightarrow \langle M_1, P\rangle|_L = \langle M_2, P\rangle|_L$$

Formally, a store M is a mapping from variables to values. The notation $M|_X$ is the restriction of the store M to the variable domain X, i.e., $M|_X = \{x \mapsto v \mid (x \mapsto v) \in M \wedge x \in X\}$. The set X^c is the complement of X. If P terminates given the initial store M, $\langle M, P\rangle$ denotes the final store; $\langle M, P\rangle = \bot$ if non-terminating.

Both definitions appear frequently with some variation in superficial details. It is easy to see that the definitions are equivalent. The second definition is particularly nice for our purpose because it is easy to see the reduction from the definition into a safety problem. Intuitively, a safety property is a property of a program which can be refuted by observing a finite trace of the program. Our definition of secure information flow only concerns the final store. Then a safety problem can be formally defined as

Definition 3 (Safety). *Let Pr be the set of all programs (for some fixed programming language). Then a safety property is a set $S \subseteq Pr$ such that there exists a logical formula $\phi(X, Y)$ such that*

$$S = \{P \mid \forall M.\langle M, P\rangle \neq \bot \Rightarrow \phi(\langle M, P\rangle, M)\}$$

A *safety problem* is a membership problem for some safety property.

Secure information flow, termination sensitive or not, is not a safety property (see, e.g., [2] for a proof). However, the termination insensitive secure information flow problem is *almost* a safety problem. To this end, we introduce the concept of a 2-*safety property* which is intuitively a property that can be refuted by observing *two* finite traces. More formally,

Definition 4 (2-Safety). *Let Pr be the set of all programs (for some fixed programming language). Then a 2-safety property is a set $S \subseteq Pr$ such that there exists a logical formula $\phi(X, Y, Z, W)$ such that*

$$S = \{P \mid \forall M_1, M_2.(\langle M_1, P\rangle \neq \bot \wedge \langle M_2, P\rangle \neq \bot) \Rightarrow \phi(\langle M_1, P\rangle, \langle M_2, P\rangle, M_1, M_2)\}$$

To distinguish, we say 1-safety when we mean safety. Clearly, any 1-safety property is a 2-safety property. The following is immediate:

Theorem 1. *The termination insensitive secure information flow problem is a 2-safety problem.*

For any program P, let $V(P)$ be the set of all variables appearing in P and let $C(P)$ be the copy of P with each $x \in V(P)$ replaced by a fresh variable $C(x)$. Any 2-safety problem can be reduced to a 1-safety problem by the following *self-composition* reduction:

Definition 5 (Self-Composition). *Let S be a 2-safety property, i.e., $S = \{P \mid \forall M_1, M_2.(\langle M_1, e\rangle \neq \bot \wedge \langle M_2, e\rangle \neq \bot) \Rightarrow \phi(\langle M_1, e\rangle, \langle M_2, e\rangle, M_1, M_2)\}$ for some ϕ. Then a self-composition reduction of S is the set*

$$\{P' \mid P' = P; C(P) \wedge \forall M_1, M_2.\langle M_1 \cup C(M_2), P'\rangle \neq \bot \Rightarrow \theta\}$$

where $\theta = \phi(\langle M_1 \cup C(M_2), P'\rangle|_{V(P)}, \langle M_1 \cup C(M_2), P'\rangle|_{V(C(P))}, M_1, M_2)$.

where the symbol ; is the sequential composition. It is easy to see that a self-composition of any 2-safety property S is a recursive subset of some 1-safety property S', i.e., given an oracle access to S', we can decide (in fact easily) if $P \in S''$ where S'' is the self-composition reduction of S. Furthermore it is easy to see that the self-composed form is equivalent to the original in the following sense:

Theorem 2. *Let S be a 2-safety property and let S' be its self-composition. Then $P \in S$ if and only if $P; C(P) \in S'$.*

Thus any 2-safety problem can be solved by reducing it to an equivalent 1-safety problem via self-composition and then solving the 1-safety problem.

In the case of the termination insensitive secure information flow problem, self-composition reduces the problem into the following problem:

Definition 6 (Secure Information Flow - Self-Composed Version). *Given a program P whose variables $H = \{h_1, \ldots, h_n\}$ are high security variables and $L = \{l_1, \ldots, l_n\}$ are low-security variables, P is said to be secure if and only if for any stores M_1 and M_2 such that $dom(M_1) = V(P)$ and $dom(M_2) = V(C(P))$ and $M_1|_{H^c} = M_2|_{C(H^c)}$,*

$$\langle M_1 \cup M_2, P; C(P)\rangle \neq \bot \Rightarrow C(\langle M_1 \cup M_2, P; C(P)\rangle|_L) = \langle M_1 \cup M_2, P; C(P)\rangle|_{C(L)}$$

where $C(M)$ is a store identical to M except that each variable x appearing in M is replaced by $C(x)$. Note that it is possible to see the above formulation directly from Definition 2 without going through the generalization of defining a 2-safety property as we have done here. As far as we know, the direct formulation appears in at least two recent papers [3, 4]. We borrowed the term "self-composition" from Barthe, D'Argenio, and Rezk [4], although they define it slightly differently.

Self-composition is a promising approach to solving difficult secure information flow instances thanks to the recent success on generic automatic software safety analysis tools such as SLAM [5] and BLAST [6], to name a few. Both SLAM and BLAST combine theorem proving and model checking in an iteratively refining manner to achieve robust safety analysis that can scale to programs of non-trivial size written in feature-rich programming languages like C. Also, they are in theory *almost* complete [7]. In practice, they have been able to verify many safety properties that were too difficult for older approaches that were not fully path-sensitive and sometimes not even flow-sensitive.

What does this progress in automatic safety analysis actually mean to secure information flow? For example, type-based information flow analysis algorithms,

```
z := 1;
if (h) then x := 1 else skip;
if (¬h) then x := z else skip;
l := x + y
```

Figure 1: The variable h is high-security and the variable l is low-security. (The variable y is not high-security.) This code is secure: regardless of the valuation of h, the low-security variable l will be $1 + y$ at the end of the program.

```
z := 1;
if (h) then x := 1 else skip;
if (¬h) then x := z else skip;
l := x + y;
z′ := 1;
if (h′) then x′ := 1 else skip;
if (¬h′) then x′ := z′ else skip
l′ := x′ + y′
```

Figure 2: Self-composition reduction applied to the program in Figure 1. For each variable x, $C(x) = x'$.

flow-sensitive or not, cannot show that the program shown in Figure 1 is secure since the low-security variable l is assigned in a branch of a conditional that depends on the high-security variable h. But a self-compositional approach can easily check that this program is secure as follows. Figure 2 is the result of applying the self-composition reduction to the program. The safety problem of whether $l = l'$ at the end of the program given $x = x' \wedge y = y' \wedge z = z' \wedge l = l'$ at the entry can be verified easily by a modern safety analysis tool. So by Theorem 2, we have automatically proved that the original program is secure. In fact, Theorem 2 implies that given a complete safety analysis, we can solve the termination insensitive secure information flow problem completely.

Before we go on to the main results of the paper, we note that it is fairly easy to carry out a similar construction for termination *sensitive* secure information flow problem by defining a "2-liveness" property which may observe up to two possibly *infinite* traces to refute the property. Self-composition can then be defined using a parallel composition instead of a sequential composition to reduce any 2-liveness problem to a 1-liveness problem. But since there are not practical frameworks for checking general software liveness properties (though some promising proposals are starting to appear [8]), we limit the content of this paper to the termination insensitive case. Also, non-deterministic programs are outside of the scope of this paper.

1.1 Contributions

The two main contributions of this paper are as follows:

– We extend the self-compositional approach to the secure information flow problem with information downgrading recently proposed by Li and Zdancewic [9].
– We identify a problem with applying the self-compositional approach in practice. We then present a solution to this problem that makes use of more traditional type-based approaches.

The first contribution was motivated by an elegant characterization of information downgrading called *relaxed non-interference* proposed recently by Li and Zdancewic [9]. Their paper contains a type-based approach for automatically checking relaxed non-interference. The self-compositional approach can in theory verify a wider range of secure programs than their type-based approach.

The second contribution starts from a disappointing discovery that the self-compositional approach, even when combined with current state-of-the-art generic automatic safety analysis tools, is too inefficient in practice. We will point out why this is the case, and offer a remedy based on previous and on-going research on type-based approaches to secure information flow, including Li and Zdancewic type system for information downgrading. The result is a framework that combines the best of both worlds, i.e., better than type-based approaches and better than the self-compositional approach.

2 Information Downgrading

"Vanilla" secure information flow as defined in Section 1 is often criticized for being too strict. For example, a security policy may permit information stored in the high-security variable *secret* to leak as long as the hash of the password from the user, say initially stored in the non-high-security variable *input*, matches with the high-security variable *hash*. For example, the following program is secure according to this policy:

$$\texttt{if } (\mathit{hashfunc}(\mathit{input}) = \mathit{hash}) \texttt{ then } l \texttt{ := } \mathit{secret} \texttt{ else skip;}$$

where l is a low-security variable. Unfortunately, the above program is not secure according to the definition of vanilla secure information flow because the valuation of l depends on the valuation of the high-security variable *secret* (and on *hash* too). In general, vanilla secure information flow does not allow *any* method of leaking *anything* about the high-security variables.

Researchers have proposed various ways to relax secure information flow to permit policies like the one above, such as robust declassification [10], delimited information release [11], and abstract non-interference [12]. A particularly nice approach called *relaxed non-interference* has been recently proposed by Li and Zdancewic [9]. Their idea is to express downgrading by the existence of a *clean function* that takes "downgraded" high-security information but does not look directly at high-security variables. Their paper is restricted to the purely functional setting, but when extended to the imperative setting, their idea can be described roughly as follows. A security policy is stated by associating each high-security variable h_i to a *downgrading function* f_{h_i}, and then we define the security of a

program P by the existence of a program $F(f_{h_1}(h_1), \ldots, f_{h_n}(h_n))$ such that F does not mention the high security variables and $F(f_{h_1}(h_1), \ldots, f_{h_n}(h_n))$ agrees with P on low-security variables at termination. Here, the notation $F(e_1, \ldots, e_n)$ refers to a program that first evaluates $e_1, \ldots, e_n$ and stores them in some variables prior to the evaluation of the rest of the program. $F(e_1, \ldots, e_n)$ can be arbitrary powerful, i.e., it need not be computable. (Readers familiar with relaxed non-interference may notice another difference – in addition to the imperative extension – from Li and Zdancewic's original definition, i.e., the use of semantic equivalence instead of syntactic equivalence rules. The consequence of this difference is discussed later in this section.) Note that secure information flow with information downgrading is more general than vanilla secure information flow; vanilla secure information flow can be expressed by setting all downgrading functions to the constant function $\lambda x.0$.

For example, in our password example, the downgrading function for *secret* can be set to

$$f = \lambda x.\texttt{if}\ (\mathit{hashfunc}(\mathit{input}) = \mathit{hash})\ \texttt{then}\ x\ \texttt{else}\ c$$

where c is some constant not in the range of values for *secret*. Then, one only needs to prove that there exists F such that $F(f(\mathit{secret}))$ is equivalent to our original program, which in this case is true by inspection. Relaxed non-interference is surprisingly general and natural. For example, it is easy to see that associating the downgrading function $\lambda x.\mathit{length}(x)$ to a secret string data implies that only the length of the string may be leaked.

We simplify the definition slightly for purpose of exposition. Formally, we use the following definition of the terminating insensitive secure information flow with information downgrading.

Definition 7 (Secure Information Flow with Information Downgrading). *Given a program P whose variables $H = \{h_1, \ldots, h_n\}$ are high security variables and $L = \{l_1, \ldots, l_n\}$ are low-security variables, P is said to be secure with respect to the downgrading policy e if and only if there exists F such that F does not mention any variable in H and for any M,*

$$\langle M, P\rangle \neq \perp \Rightarrow (\langle M, F(e)\rangle \neq \perp \wedge \langle M, P\rangle|_L = \langle M, F(e)\rangle|_L)$$

Here, e is any side-effect free expression. It is easy to see that our definition is at least as expressive as Li and Zdancewic style of using explicit downgrading functions. For example, vanilla secure information flow can be obtained by setting e to be the tuple $(h_1, \ldots, h_n)$. For the password example, e is

$$\texttt{if}\ (\mathit{hashfunc}(\mathit{input}) = \mathit{hash})\ \texttt{then}\ \mathit{secret}\ \texttt{else}\ c$$

It is worth pointing out that the above definition is slightly different from that of Li and Zdancewic's since we use semantic equivalence to check that $\langle M, P\rangle|_L = \langle M, F(e)\rangle|_L$ whereas Li and Zdancewic take a less complete (but still sound) equivalence relation as the definition. Their paper contains a discussion on why a weaker equivalence may be desirable in some situations. However,

it is not clear whether using a weaker equivalence based on intentional syntactic equivalence rules as done in their paper is best. Perhaps a more principled approach is to equate some computational hardness properties as well as semantic equivalence. For example, any $F(\lambda x.\texttt{if } (\mathit{password} = x) \texttt{ then } 1 \texttt{ else } 0)$ semantically equivalent to $l := \mathit{password}$ on the variable l will be computationally expensive assuming that the set of valuations of *password* is large. Note that there is an F such that semantic equivalence alone will not be able to distinguish $F(\lambda x.\texttt{if } (\mathit{password} = x) \texttt{ then } 1 \texttt{ else } 0)$ from $l := \mathit{password}$, namely the one that tries all possible strings. In this paper, we stick with semantic equivalence.

We now prove the following.

Theorem 3. *The termination insensitive secure information flow with information downgrading is a 2-safety problem.*

The formal proof appears in our companion technical report [13]. The proof establishes the equivalence of Definition 7 to the following predicate

$$\forall M_1, M_2.(\langle M_1, P\rangle \neq \bot \wedge \langle M_2, P\rangle \neq \bot) \Rightarrow$$
$$((M_1|_{H^c} = M_2|_{H^c} \wedge \langle M_1, e\rangle = \langle M_2, e\rangle) \Rightarrow \langle M_1, P\rangle|_L = \langle M_2, P\rangle|_L)$$

The predicate is actually equivalent to the definition of *delimited information release* [11] restricted to the safety case. Therefore, the above proof shows that relaxed non-interference with semantic equivalence is roughly (modulo the imperative extension) equivalent to that of delimited information release. Since Barthe, D'Argenio, and Rezk [4]'s formulation of self-composition is flexible enough to handle delimited information release, our result also shows that their framework can be used as a black box to solve secure information flow problems with information downgrading in the style of relaxed non-interference.

Concretely, self-composition reduces the termination insensitive secure information flow with information downgrading to the following problem:

Definition 8 (Secure Information Flow with Information Downgrading - Self-Composed Version). *Given a program P whose variables $H = \{h_1, \ldots, h_n\}$ are high security variables and $L = \{l_1, \ldots, l_n\}$ are low-security variables, P is said to be secure with respect to the downgrading policy e if and only if for any stores M_1 and M_2 such that $dom(M_1) = V(P)$ and $dom(M_2) = V(C(P))$, $M_1|_{H^c} = M_2|_{C(H^c)}$, and $\langle M_1, e\rangle = \langle M_2, C(e)\rangle$,*

$$\langle M_1 \cup M_2, P; C(P)\rangle \neq \bot \Rightarrow C(\langle M_1 \cup M_2, P; C(P)\rangle|_L) = \langle M_1 \cup M_2, P; C(P)\rangle|_{C(L)}$$

As in the case of vanilla secure information flow, this self-compositional reduction is complete. Hence in theory, a complete safety analysis can decide any instance of the problem. In practice, the self-compositional approach can check cases where Li and Zdancewic's type-based approach would fail. For example, the program in Figure 3 is secure according the downgrading policy $\texttt{if } (\mathit{hashfunc}(\mathit{input}) = \mathit{hash}) \texttt{ then } \mathit{secret} \texttt{ else } c$. Essentially, the program is same as our original example except that we have added a few small things so that the code isn't exactly like the downgrading policy. The program can be

```
if (hashfunc(input) = hash) then
   t := t + 1; l := l + secret
else skip
```

Figure 3: The variables *secret* and *hash* are high-security and the variable *input* and l are low-security. This code is secure according to the downgrading policy `if` (*hashfunc*(*input*) = *hash*) `then` *secret* `else` 0.

easily proved to be secure via the self-compositional approach; the downgrading policy leads to a conditional predicate, but that is no harder than handling conditionals in the program body, and therefore a path-sensitive safety analysis can quickly check that the safety property is satisfied in the self-composed program (not shown). On the other hand, conventional type-based approaches would break in the presence of these small changes since they are more dependent on the structure of downgrading operations.

3 Self-composition in practice, its problem, and a solution

The main appeal of the self-compositional approach to secure information flow comes from the recent successes with automatic safety analysis tools in verifying a very broad range of safety properties in real programs, including ones that are path-sensitive, flow-sensitive, and (linear) arithmetic sensitive. Furthermore, automatic safety property checking is an active area of research with frequent improvements, and therefore even if some self-composed instances of a secure information flow problem cannot be solved by the existing tools today, it may not be unreasonable to expect them to be solved by the next generation of safety analysis tools. That is, the self-compositional approach automatically benefits from improvements to the underlying safety analysis. Furthermore, the self-compositional approach needs nothing more than off-the-shelf tools, and so it has an engineering advantage over type-based approaches.

In this section, we argue that such an optimistic prospect is unrealistic in practice. 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 in safety analysis designed for "natural" safety problems (i.e., ones that are naturally 1-safety) to be able to close the gap significantly.

We first motivate our argument by a simple example. Figure 4 is a program which computes the nth Fibonacci number and sets the low-security variable l to 1 if the nth Fibonacci number is greater than k and to 0 otherwise. The program contains no high-security variables, so it is trivially secure. Let us apply the self-composition reduction by renaming each variable x to x' in the copy (shown in Figure 5). We would like the safety analysis tools to check that $l = l'$ at the end of the program provided that for each variable x in the original, $x = x'$ at

```
while (n > 0) do
   f1 := f1 + f2; f2 := f1 - f2; n := n - 1;
if (f1 > k) then l := 1 else l := 0;
```

Figure 4: The while loop computes the nth Fibonacci number. The variable l is low security, which is set to 1 if the nth Fibonacci number is greater than k, and is set to 0 otherwise. There are no high-security variables.

```
while (n > 0) do
   f1 := f1 + f2; f2 := f1 - f2; n := n - 1;
if (f1 > k) then l := 1 else l := 0;
while (n' > 0) do
   f1' := f1' + f2'; f2' := f1' - f2'; n' := n' - 1;
if (f1' > k') then l' := 1 else l' := 0;
```

Figure 5: The program in Figure 4 after self-composition.

the beginning of the program. However, a state-of-the-art safety analysis tool BLAST [6] fails to terminate given this query; more precisely, BLAST endlessly keeps discovering more and more predicates getting closer and closer to the answer but never actually converging.[3]

Why does this happen? The reason is that the modern generic safety analysis tools gain their robustness by moving away from structure-dependent reasoning and instead trying to solve the problem semantically. In the case above, if BLAST could verify that $l = l'$ at the end of the self-composed program, then that roughly means that it was able to show that the upper part of the original code was computing a Fibonacci number for each n. We believe that this problem also applies to other safety analysis tools for imperative languages based on a Hoare-style reasoning framework since the framework encourages verifying a property about the whole program by locally reasoning about the store update at each statement. We give more details supporting this argument in Section 3.1.

Even if BLAST was improved with more arithmetic-related reasoning power or if we used another tool that can verify the correctness of our Fibonacci computation loop, there would be always another example whose partial correctness would be too difficult for the tool to verify automatically. Why does this matter to the self-compositional approach to secure information flow? Because there are many programs that compute arbitrary values in complex ways, and it is fair to expect that these values can flow to low-security variables since the low-security variables are the observable outputs of the program. (On the other hand, parts of the program where high-security values flow can be expected to be small and not too complex in most real security-aware applications.)

[3] We used the latest version (as of March 2005) obtained directly from the BLAST group.

$$P ::= x := e \mid \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \mid \texttt{while}\ e\ \texttt{do}\ P \mid P_1; P_2 \mid \texttt{skip}$$

Figure 6: The syntax of `While`. e is some reasonable expression such as integer arithmetics, comparisons, and boolean operations.

$$\begin{aligned}\varepsilon ::= {} & [\,] \mid x := \varepsilon \mid \texttt{if}\ \varepsilon\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \mid \texttt{if}\ e\ \texttt{then}\ \varepsilon\ \texttt{else}\ P \mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ \varepsilon \mid \\ & \texttt{while}\ \varepsilon\ \texttt{do}\ P \mid \texttt{while}\ e\ \texttt{do}\ \varepsilon \mid \varepsilon; P \mid P; \varepsilon\end{aligned}$$

Figure 7: The contexts of `While`.

Therefore, what the self-compositional approach needs is some reasoning extension that can make use of the inherent symmetry and redundancy in self-composed programs but not in ordinary programs. For example, in the case of the Fibonacci program, this reasoning extension should be able to tell that the loops are equivalent by the fact that both loops are just copies of the same code with each copied variable in the code starting with the same value as the original. On the other hand, if copies of some code actually use variables with different initial values, then this reasoning system should safely say that "I do not know if they are equivalent" so that a more powerful reasoning logic can work out the details.

Such a reasoning extension is exactly where type-based approaches to secure information flow excel. That is, the "same value variables" are the low security variables, and "different value variables" are the high-security variables. Indeed, type-based approaches can easily verify our Fibonacci program by carrying out roughly the following logical reasoning: f_1 is only assigned low-security values in a while loop with a low-security guard, and hence l is assigned only in a conditional statement of a low-security condition which implies that l is low-security. But as we have seen in the previous sections, there are instances of secure information flow that cannot be verified by type-based approaches but can be easily verified by the self-compositional approach. To this end, we generalize this line of thought to design an approach to secure information flow that combines the best parts of the two approaches.

3.1 Type-directed transformation for secure information flow

We illustrate our idea using the imperative language `While` defined in Figure 6. The semantics of `While` is completely standard. While we choose this simple language for purpose of exposition, it is not hard to adapt our approach to more complex languages.

To motivate the idea, consider the program $P = \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$. If a secure information flow type system gives e a low-security type, then the self composition $P; C(P)$ is equivalent to the program $\texttt{if}\ e\ \texttt{then}\ (P_1; C(P_1))\ \texttt{else}\ (P_2; C(P_2))$

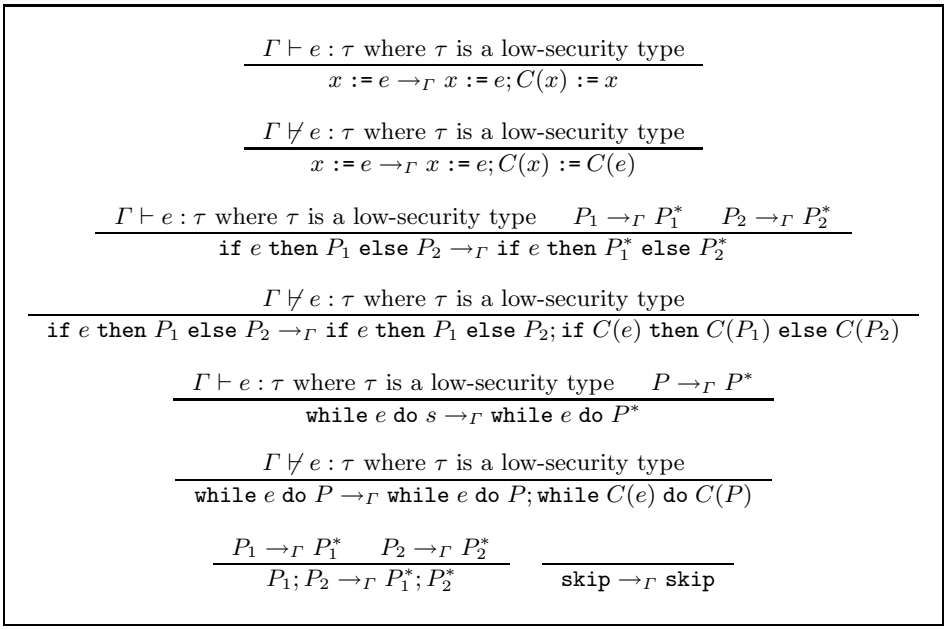

Figure 8: Type-directed translation $\rightarrow_\Gamma$. "$\Gamma \nvdash e : \tau$ where τ is a low-security type" means that $\Gamma \vdash e : \tau$ is not derivable for any low-security type τ.

provided that the values of the low-security variables between the original and the copy are equal at the beginning of the program. Now, suppose that e is (or was the result of) a complex computation like our Fibonacci loop. Then using the second form instead of $P; C(P)$, a safety analysis tool is able to bypass checking whether e is equal to $C(e)$ without losing precision or efficiency. Furthermore, we may recursively apply the same idea to P_1 and P_2 so that we may not even need to use $C(P_1)$ and $C(P_2)$.

We now generalize this idea to design a *type-directed transformation* for secure information flow. To this end, we first define the contexts ε of `While` in a completely standard manner given in Figure 7. Our type-directed transformation is parametrized by a secure information flow type system. Rather than defining a type-directed transformation for each different type system and proving the correctness each time, we formally state what our type-directed transformation expects from a secure information flow type system so that we can design one type-directed transformation for all type systems satisfying the definition and prove its correctness once and for all.

Definition 9. *Given a secure information flow problem with information downgrading problem instance (P, H, L, e) (see Definition 7), secure information flow type inference is an algorithm that outputs a type environment Γ with the relation $\sim_\Gamma$ satisfying all of the following.*

(1) For any M_1 and M_2, if $M_1|_{H^e} = M_2|_{H^e}$ and $\langle M_1, e\rangle = \langle M_2, e\rangle$ then $M_1 \sim_\Gamma M_2$.

(2) For any P such that $\Gamma \vdash P$ and for any M_1 and M_2 such that $M_1 \sim_\Gamma M_2$, $\langle M_1, P\rangle \sim_\Gamma \langle M_2, P\rangle$.
(3) For any e such that $\Gamma \vdash e : \tau$ and τ is a low-security type, for any M_1 and M_2 such that $M_1 \sim_\Gamma M_2$, $\langle M_1, e\rangle = \langle M_2, e\rangle$.
(4) For any ε and P, if $\Gamma \vdash \varepsilon[P]$, then $\Gamma \vdash P$.
(5) $\Gamma \vdash P$

Intuitively, the first condition says that the precondition of the original security policy is at least as strong as the relation $\sim_\Gamma$. The second condition says that $\sim_\Gamma$ is preserved by the program semantics. The third condition says that if an expression is typed with a low-security type, then it in fact is low-security with respect to $\sim_\Gamma$. The fourth condition is a standard structural property for (flow-insensitive) type systems. The last condition says that P itself can be typed under Γ.

For example, the well-known Volpano and Smith type inference algorithm [14] when restricted to the language `While` can satisfy the above requirement for vanilla secure information flow (i.e., the downgrading policy e is some constant) by letting

$$\sim_\Gamma = \{(M_1, M_2) \mid M_1(x) = M_2(x), x{:}\tau \in \Gamma \text{ where } \tau \text{ is a low-security type}\}$$

Defining $\sim_\Gamma$ for Li and Zdancewic type system [9] (when adapted to the language `While` in a natural way) is also not difficult:

$$\sim_\Gamma = \{(M_1, M_2) \mid \langle M_1, e\rangle = \langle M_2, e\rangle, \Gamma \vdash e : \tau \text{ where } \tau \text{ is a low-security type}\}$$

(Indeed, this definition, also works for the Volpano and Smith type system although it is unnecessarily more elaborate than the one above. This fact is not surprising since Li and Zdancewic system can be thought of as an extension to the Volpano and Smith system.) Due to space constraints, we do not formally describe any specific type inference algorithm in this paper and instead ask readers to refer to the cited references. Our companion technical report discusses how to adapt our approach to secure information flow type systems that do not quite meet these requirements [13].

It is important to note that we do not need an algorithm that actually computes the relation $\sim_\Gamma$. Instead, merely the existence of such a relation is enough since $\sim_\Gamma$ is only used explicitly when proving the correctness of the type-directed transformation.

We now describe our type-directed transformation. Given a problem instance (P, H, L, e) and Γ produced by the corresponding secure information flow type inference, the type-directed transformation $\rightarrow_\Gamma$ is defined by the rules shown in Figure 8. In order to solve the given problem instance, we first apply this transformation to P to obtain a program P^*, i.e., $P \rightarrow_\Gamma P^*$. Then we ask a safety analysis tool whether for any M_1 and M_2 such that $dom(M_1) = V(P)$, $dom(M_2) = V(C(P))$, $M_1|_{H^c} = M_2|_{C(H^c)}$, and $\langle M_1, e\rangle = \langle M_2, C(e)\rangle$, whether

$$\langle M_1 \cup M_2, P^*\rangle \neq \perp \Rightarrow C(\langle M_1 \cup M_2, P^*\rangle|_L) = \langle M_1 \cup M_2, P^*\rangle|_{C(L)}$$

```
while (n > 0) do
   f1 := f1 + f2; f2 := f1 - f2; n := n - 1;
if (h) then x := 1 else skip;
if (¬h) then x := 1 else skip;
while (i < f1) do
   l := l + x; i := i + 1
```

Figure 9: The variable h is high-security and the variable l is low-security. The program is secure but cannot be verified by either a type-based approach or self-composition.

```
while (n > 0) do
   f1 := f1 + f2; f1' := f1; f2 := f1 - f2; f2' := f2;
   n := n - 1; n' := n;
if (h) then x := 1 else skip; if (h') then x' := 1 else skip;
if (¬h) then x := 1 else skip; if (¬h') then x' := 1 else skip;
while (i < f1) do
   l := l + x; l' := l' + x'; i := i + 1; i' := i
```

Figure 10: The program from Figure 9 after the type-directed transformation.

That is, we ask the same query as the self-compositional approach except that we use P^* in place of $P; C(P)$.

As an example, consider the program shown in Figure 9. The program exhibits interactions of features discussed in previous sections that made type-based approaches and the self-composition approach fail (at least when using BLAST as the underlying safety analysis). Therefore, it can be checked by neither method. Applying the type-directed transformation using Volpano and Smith type inference algorithm, we obtain the program P^* shown in Figure 10. Note that both loop conditions remain unduplicated (though their bodies are duplicated) since both conditions can be given low-security types. BLAST can easily decide that $l = l'$ at the end of P^* provided that $n = n'$, $f_1 = f_1'$, $f_2 = f_2'$, $i = i'$, and $l = l'$ at the beginning, i.e., it can prove that the program is secure. In fact, BLAST is clever enough that it will not even bother to look carefully at the first loop (which was the part that made BLAST fail in the self-composition approach!) since it quickly notices simply by looking at the code following the loop that it can prove $l = l'$ at the end of the program regardless of what values are stored in f_1, f_1', f_2', n, and n' after the loop.

We now prove the correctness of the type-directed transformation approach. The following lemma is the main technical result.

Lemma 1. *Suppose $P \rightarrow_\Gamma P^*$ where Γ is the output of a secure information flow type system given (P, H, L, e) satisfying Definition 9. Then, for any M_1 and M_2 such that $M_1 \sim_\Gamma M_2$, if $\langle M_1, P\rangle \neq \bot$ and $\langle M_2, P\rangle \neq \bot$ then*

$$\langle M_1, P\rangle = \langle M_1 \cup C(M_2), P^*\rangle|_{V(P)} \wedge C(\langle M_2, P\rangle) = \langle M_1 \cup C(M_2), P^*\rangle|_{V(C(P))}$$

The proof appears in our companion technical report [13].

Theorem 4. *For any M_1 and M_2 such that $M_1|_{H^c} = M_2|_{H^c}$, $\langle M_1, e\rangle = \langle M_2, e\rangle$, $\langle M_1, P\rangle \neq \bot$, and $\langle M_2, P\rangle \neq \bot$*

$$\langle M_1, P\rangle|_L = \langle M_2, P\rangle|_{C(L)} \longleftrightarrow C(\langle M_1 \cup C(M_2), P^*\rangle|_L) = \langle M_1 \cup C(M_2), P^*\rangle|_{C(L)}$$

where $P \rightarrow_\Gamma P^*$ and Γ is the output of a secure information flow type system given (P, H, L, e) satisfying Definition 9.

Proof. Immediate from condition (1) in Definition 9 and Lemma 1.

Therefore the type-directed transformation approach is sound and complete up to the soundness and completeness of the underlying safety analysis.

The type-directed transformation is inexpensive relative to the complexity of the underlying type inference algorithm. It is easy to see that for $P \rightarrow_\Gamma P^*$, the size of P^* is at most two times the size of P. Computing P^* from P takes time linear in P and the number of $\Gamma \vdash e : \tau$ queries made to the type inference algorithm. However, most secure information flow type systems actually compute the principal types for each expression. In such a case, asking whether there is a low-security type τ such that $\Gamma \vdash e : \tau$ is a constant time operation once the principal types have been computed for P.

It is clear that the type-directed transformation approach is better than a type-based approach alone since it runs the type inference algorithm as a subprocess, and therefore it may accept the program if the type inference successfully assigned low-security types to the low-security variables.

Before we argue that the type-directed transformation approach is better than the self-compositional approach, we point out that in their full generality, the two approaches are equivalent since they are both a complete characterization of the same secure information flow problem, i.e., they are no different to a hypothetical safety analysis having infinite deduction power. Even restricted to the class of safety analysis tools that are "fast" and sound (but not necessarily complete), we cannot compare the two because, for example, this class includes one that rejects all programs not of the form $P; C(P)$, i.e., the self-composition approach is always better for such a safety analysis, and conversely, there is a sound safety analysis that rejects all programs of the form $P; C(P)$.

Instead, we argue that type-directed-transformed programs tend to be more digestible than self-composed programs for most automatic safety analysis tools assuming that they are targeted toward the general class of "natural" safety (i.e., naturally 1-safety) problems for imperative languages. Such tools typically reason about a program by interpreting each program statement as an abstract store update operation where an abstract store may be a set of abstract values stored in abstract memory cells, a set of predicates over program variables where each predicate represents a possible store, or something similar. With self-composition, the store space for the copies P and $C(P)$ are completely disjoint. However, the query is all about connecting these two stores, i.e., it is about

whether some portion of the two disjoint store spaces is equivalent after the program terminates given that some portion of the two disjoint store spaces is equivalent before the program. Therefore 1-safety analysis tools generally suffer from not being able to relate the two stores within the abstract interpretation phase. Our type-directed transformation directly makes *relevant* connections between the two stores locally within the program. These connections help the safety analysis significantly in some situations as seen in the example in this section (Figure 9, 10) where the self-compositional approach would perform poorly.

4 Related Work

Darvas, Hähnle, and Sands [3] used a self-compositional approach to prove secure information flow properties for Java CARD programs. They used an interactive approach instead of an automatic approach. Barthe, D'Argenio, and Rezk coined the term "self-composition" in their paper [4]. Their paper is mostly theoretical results on characterizing various secure information flow problems, including non-deterministic and termination-sensitive cases, in a self-compositional framework. We believe that our paper is the first one to examine applying an automatic safety analysis in the self-compositional setting.

Barthe, D'Argenio, and Rezk in the same paper showed that their self-compositional framework can handle delimited information release as originally proposed by Sabelfeld and Myers [11]. We have shown that Li and Zdancewic's recently proposed relaxed non-interference [9] is equivalent to delimited information release when strengthened with semantic equivalence. Relaxed non-interference is arguably a more natural formulation of information downgrading than delimited information release. Our paper suggests a promising practical approach toward making complete use of properties definable as relaxed non-interference.

5 Conclusions and future work

We have shown that Li and Zdancewic's relaxed non-interference can be incorporated into both self-composition and its generalization, the type-directed transformation approach. We have presented the type-directed transformation approach as a solution to a problem with applying self-composition in practice with off-the-shelf automatic safety analysis tools. The type-directed transformation approach combines the best parts of traditional type-based approaches and self-composition.

One possible improvement to our type-directed transformation is to make it iterative, i.e., in the event that the safety analysis fails, instead of failing the whole process completely it may report back to the type system with information about which expressions are low-security at which program points. Then the type system can "cast" these expressions to low-security types to obtain more low-security expressions, and the process repeats. To make this work, we need a way to obtain partial results from the safety analysis tool. Obtaining useful partial results may be difficult for a demand-driven framework such as BLAST.

References

1. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications **21** (2003) 5–19
2. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: SP '94: Proceedings of the 1994 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society (1994) 79
3. Darvas Á, Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In Gorrieri, R., ed.: Workshop on Issues in the Theory of Security, WITS, IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS (2003)
4. Barthe, G., D'Argenio, P., Rezk, T.: Secure information flow by self-composition. In: Computer Security Fundation Workshop (CSFW'17), IEEE Press (2004)
5. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon (2002) 1–3
6. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon (2002) 58–70
7. Ball, T., Podelski, A., Rajamani, S.K.: Relative completeness of abstraction refinement for software model checking. In Kaoen, J.P., Stevens, P., eds.: Proceedings of TACAS02: Tools and Algorithms for the Construction and Analysis of Systems. Volume 2280 of LNCS., Springer-Verlag (2002) 158–172
8. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Long Beach, California (2005) 132–144
9. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Long Beach, California (2005) 158–170
10. Zdancewic, S., Myers, A.C.: Robust declassification. In: CSFW '01: Proceedings of the 14th IEEE Workshop on Computer Security Foundations, IEEE Computer Society (2001) 15–23
11. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Proceedings of the International Symposium on Software Security (ISSS'03). (2003)
12. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: parameterizing non-interference by abstract interpretation. In: Proceedings of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, Italy (2004) 186–197
13. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. University of California, Berkeley UCB//CSD-05-1396 (Technical report)
14. Volpano, D., Smith, G.: A type-based approach to program security. In Bidoit, M., Dauchet, M., eds.: Theory and Practice of Software Development, 7th International Joint Conference. Volume 1214 of Lecture Notes in Computer Science., Lille, France, Springer-Verlag (1997) 607–621