An Analysis of Universal Information Flow based on Self-Composition

Christian Müller, Máté Kovács and Helmut Seidl
Fakultät für Informatik
TU München
Munich, Germany
Email: {muellchr,kovacsm,seidl}@in.tum.de

Abstract—We introduce a novel way of proving information flow properties of a program based on its self-composition. Similarly to the universal information flow type system of Hunt and Sands, our analysis explicitly computes the dependencies of variables in the final state on variables in the initial state. Accordingly, the analysis result is independent of specific information flow lattices, and allows to derive information flow w.r.t. any of these. While our analysis runs in polynomial time, we prove that it never loses precision against the type system of Hunt and Sands, and may gain extra precision by taking similarities between different branches of conditionals into account. Also, we indicate how it can be smoothly generalized to an interprocedural analysis.

Index Terms—information flow control; hypersafety property; noninterference; weakest precondition; interprocedural analysis

I. INTRODUCTION

In order to enforce confidentiality in application systems which are accessed by multiple principals, it does not suffice to distinguish two security levels only. Therefore, Denning and Denning provide the concept of information flow lattices and provide an algorithm to certify the conformance of structured programs with a given information flow policy [9], [10]. This algorithm later has been formalized as a type system and proven correct w.r.t. the operational semantics [20]. Subsequently, a series of papers has extended this type system to cope, e.g., with object oriented programming languages [16], with multi-threaded programs [22], and also to enable richer policy specification languages [21], [15].

While the original certification by Denning and Denning was flow-insensitive, later analyses provide methods for taking the flow of control within programs into account. In particular, this is the case by the JOANA system, [18], which is based on program dependence graphs [19]. Interestingly, it turns out that the same program need not be re-evaluated if several information flow lattices are of concern. Hunt and Sands observe that there is a universal information flow lattice from which all other information flow properties can be inferred [11]. This lattice is given by the powerset of the set of variables occurring in the program, where a flow-sensitive interprocedural information flow analysis for this lattice is provided in [13].

In order to deal with implicit information flows, the analysis, e.g., of Hunt and Sands assigns security levels to reached program points which in turn may affect the security levels of variables modified at these program points. In some cases, though, this approach may lead to unnecessary loss in precision.

Example 1. Consider the following program:

\[
\begin{align*}
\text{\texttt{y} } &\leftarrow\ 0; \\
\text{\texttt{if} (secret = 0) } &\{ \\
\text{\texttt{x} } &\leftarrow\ 0; \\
\text{\texttt{y} } &\leftarrow\ y + 1; \\
\text{\texttt{else} } &\{ \\
\text{\texttt{x} } &\leftarrow\ 0; \\
\text{\texttt{}}&\}
\end{align*}
\]

This program consists of a branching construct, where the branching condition depends on the secret. Flow-sensitive type systems such as [13] as well as methods based on program dependence graphs, would conclude that the final value of the variable \(x\) may also depend on the initial value of the variable secret, since it is assigned to inside the branching construct. A more detailed analysis, however, may take into account that both branches affect the variable \(x\) in the same way (even though they behave differently w.r.t. to \(y\)), and therefore omit the dependency of \(x\) on the variable secret.

This example may seem contrived, as it could be handled by existing methods if only the duplicated assignment \(x \leftarrow 0\) had been moved behind the conditional before-hand. This preprocessing may, however, not always be as easily possible, e.g., when the matching program parts are more subtly intertwined or wrapped into distinct procedure calls (see Example 8).

The loss in precision thus can already be observed for two security levels only, namely, high and low. The property that publicly available data produced by the program may not depend on secret input, has also been called noninterference [25]. For proving noninterference of imperative deterministic programs, recently alternative methods have been proposed, which are more directly based on a formulation of noninterference as a hypersafety property [1], [5], [7]. Conceptually, proving noninterference compares any pair of executions of the given program which differ only in secret values for certain variables in the initial state. Noninterference for a variable \(x\) at program exit is guaranteed if the values for \(x\) provided by both executions are always equal. In [5], a calculus of Hoare-like rules for pairs of programs is provided. A similar approach
[y ← y + 1, y ← y + 1];
if (secret = 0, secret = 0) {
  [x ← 0, x ← 0];
  [y ← y + 1, y ← y + 1];
} else {
  if (~secret = 0, secret = 0) {
    [x ← 0, x ← 0];
    [skip, y ← y + 1];
  } else {
    if (secret = 0, ~secret = 0) {
      [x ← 0, x ← 0];
      [y ← y + 1, skip];
    } else {
      [x ← 0, x ← 0];
    }
  }
}

Fig. 1. Example Self-composition

is followed by Nanevski et al. [1] who provide a relational Hoare type theory inside Coq for an interactive verification of this property. The fully automated approach of [7], [24], on the other hand, proceeds in two separate phases. In the first phase, a self-composition of the program is constructed, which represents all pairs of executions of the original program on distinct copies of the program state. In the second phase, this program is then analyzed for equality of corresponding variables by means of relational abstract interpretation.

**Example 2.** Consider again the program from Example 1. A possible corresponding self-composition is shown in Figure 1.

Operations which are aligned by the self-composition are put into square brackets. The same holds true for conditions where both conditions in a pair must evaluate to true for the corresponding branch of the if statement to be taken. Since in all branches of the self-composed program, the variable \( x \) is assigned the same value, they must be necessarily equal at program exit — no matter which values for secret have been chosen.

Self-compositions which attempt to align similar program parts can, e.g., be computed efficiently by means of syntactic matching of syntax trees [7]. So far, the analyses based on self-composition have only dealt with two security levels, namely, secret and public (i.e., high and low), whereas the type-based approaches naturally can also deal with more refined security lattices in the sense of [9] which allow for more refined confidentiality policies. Here, we show that a universal information flow analysis can also be constructed via self-composition. For that, we provide a formulation of universal information flow based on a calculus of preconditions for a suitable self-composition of the program. As demonstrated by Example 2, the result is sometimes more precise than the result computed by the methods for universal information flow analysis in [11], [13]. For a comparison, we prove that it is always at least as precise and so yields strictly better or equal results.

The immediate formulation of our calculus relies on Boolean combinations of equality assertions — implying that the resulting algorithm runs in exponential time. Subsequently, we provide a non-trivial reformulation of the calculus which requires to track conjunctions of assertions of equalities between program variables only — implying that the resulting algorithm runs in polynomial time. Finally, we indicate how our methods can be extended to deal with possibly recursive procedures as well.

The paper is organized as follows. In Section II, we introduce our basic notion of programs and self-compositions of programs. In Section III, we present a weakest precondition formulation for inferring equalities of two copies of the same variable w.r.t. pairs of programs. In Section IV, we show that conjunctions of variable equalities are sufficient to realize this analysis. In Section V, we indicate in which sense our calculus realizes an analysis of universal information flow. In Section VI, our calculus is then compared to the type-based analysis of flow-sensitive universal information flow according to [14], [11], [13] and shown to be at least as precise. In Section VII, the calculus is generalized to an interprocedural analysis of flow-sensitive universal information flow. Finally, we compare our results to the work of other authors in Section VIII.

## II. SELF-COMPOSITIONS OF PROGRAMS

In this paper we consider programs \( p \) given by the following grammar:

\[
\begin{align*}
\text{(program)} & : = \text{def}_1 \ldots \text{def}_k \\
\text{(procedures)} & : = \text{def}\{\text{stmts}\} \\
\text{(statements)} & : = \text{stmt}_1 \ldots \text{stmt}_n \\
\text{(statement)} & : = x \leftarrow e; | f(); | \text{skip} \\
& \quad \text{if } (c) \{\text{stmts}\} \text{ else } \{\text{stmts}\} | \text{while } (c) \{\text{stmts}\}
\end{align*}
\]

A program \( p \) consists of a sequence of procedure definitions where program execution starts with the call to a dedicated procedure main. A procedure definition def consists of a name and a body, which is a list of statements. A statement is either an assignment \( (x \leftarrow e) \), a conditional if, or a while-loop. The symbols e and c denote expressions which are built up from variables and operators. For simplicity, explicit declarations of variables have been omitted. Instead, programs operate on a finite set \( G \) of global variables.

For programs, we consider a big-step operational semantics along the lines of [23]. For a program fragment \( p \), and variable assignments \( \sigma, \sigma' \), the triple \( p : \sigma \rightsquigarrow \sigma' \) denotes that the execution of program \( p \) on the initial state \( \sigma \) terminates with the final state \( \sigma' \). The rules for defining this relation for our minimalist language are shown in Figure 2, where \( \sigma[e] \) denotes the value returned by the evaluation of the expression e w.r.t. the variable assignment \( \sigma \).
In [7], [24], a composition operation \([\cdot, \cdot]\) of programming constructs is presented resulting in a 2-program. Intuitively, a 2-program \(pp\) operates on pairs of states as considered by an ordinary program \(p\). Instead of assignments of program variables, a 2-program has pairs of aligned assignments as basic operations each referring to the corresponding component. Thus, the aligned assignment \([x \leftarrow x + 1, x \leftarrow x + 1]\) simultaneously increments the variable \(x\) in both components of the current state. Assignments of the form \(x \leftarrow x\) do not modify the corresponding component and therefore are also denoted by \(\text{skip}\). Likewise, 2-programs use aligned control structures such as aligned conditionals and aligned loops. The conditions of these control-structures consist of pairs \((c_1, c_2)\) of ordinary conditions \(c_1, c_2\) where \(c_1\) and \(c_2\) refer only to variables of the first or second component, respectively. The understanding is that the pair \([c_1, c_2]\) evaluates to \(\text{tt}\) iff both \(c_1\) and \(c_2\) evaluate to \(\text{tt}\) for their respective component of the state. 2-programs may also contain aligned procedure calls \([f_1(), f_2()]\) where again \(f_i\) operates on the \(i\)-th component of the program state only. The semantics of a 2-program can be described by a relation \(pp: (\sigma, \tau) \rightarrow (\sigma', \tau')\) where for the 2-program \(pp\) resulting from the self-composition \([p, p]\) we have:

\[
\text{(S) } pp: (\sigma, \tau) \rightarrow (\sigma', \tau') \text{ iff } p: \sigma \rightarrow \tau \text{ and } p: \tau \rightarrow \tau'.
\]

Here, we will not repeat the technical details of the specific composition \([\cdot, \cdot]\) of [7], [24]. Rather, we present reasonable requirements, met by the construction there, for an operation \([\cdot, \cdot]\) so that the 2-program resulting from \([p, p]\) satisfies (S). These requirements are:

**Composition with skip.** For any statement \(s\), the compositions \([s, \text{skip}]\) and \([\text{skip}, s]\) modify one state according to \(s\) and leave the other intact, we have:

\[
[s_1 \ldots s_k, \text{skip}] = [s_1, \text{skip}] \ldots [s_k, \text{skip}]
\]

\[
[\text{if} (b) \{p_1\} \text{ else } \{p_2\}, \text{skip}] = \begin{cases} 
[\text{if} (b, \text{tt}) \{ 
[p_1, \text{skip}] 
\} \text{ else } \{ 
[p_2, \text{skip}] 
\} 
\end{cases}
\]

\[
\text{if} (c) \{p_1\} \text{ else } \{p_2\}: \sigma \rightarrow \tau \text{ if } \sigma \models \neg c
\]

\[
p: \sigma \rightarrow \sigma' \text{ while } (c) \{p\}: \sigma' \rightarrow \tau \text{ if } \sigma \models c
\]

\[
\text{while } (c) \{p\}: \sigma \rightarrow \tau \text{ if } \sigma \models \neg c
\]

\[
p: \sigma \rightarrow \tau \text{ if } p \text{ is the body of } f
\]

\[
[f(), \text{skip}] = [p, \text{skip}]
\]

Here we assume that procedure \(f\) has body \(p\). The rules for compositions \([\text{skip}, s]\) are analogous.

**Identical sequences.** For a sequence of statements \(s_1 \ldots s_n\) we have:

\[
[s_1 \ldots s_n, s_1 \ldots s_n] = [s_1, s_1] \ldots [s_n, s_n]
\]

**Non-identical sequences.** If the sequences \(s_1 \ldots s_m\) and \(s'_1 \ldots s'_n\) are not identical, then

\[
[s_1 \ldots s_m, s'_1 \ldots s'_n] = [t_1, t'_1] \ldots [t_r, t'_r]
\]

where the sequences \(s_1 \ldots s_m\) and \(t_1 \ldots t_r\) as well as the sequences \(s'_1 \ldots s'_n\) and \(t'_1 \ldots t'_r\) coincide — up to insertions of \(\text{skip}\) instructions into the sequence. Moreover, for every \(i = 1, \ldots , r\), the pair \(t_i, t'_i\) is composable. Here, we call two statements \(t, t'\) composable if at least one of them equals \(\text{skip}\), or one of the following properties holds:

- \(t, t'\) are syntactically identical assignments;
- both are procedure calls;
- both are \(\text{if}\)-statements with identical conditions; or
- both are \(\text{while}\)-statements with identical conditions.

These assumptions can be met by multiple realizations of the composition operation, which may differ in the strategy how the statements in sequences are aligned. Possible heuristics for constructing decent alignments can, e.g., be found in [7], [24]. One trivial possibility is to compose all statements in the two sequences with \(\text{skip}\) and then to concatenate the respective results as in:

\[
[s_1 \ldots s_m, s'_1 \ldots s'_n] = [s_1 \ldots s_m, \text{skip}] [\text{skip}, s'_1 \ldots s'_n]
\]

In this case however, the potential similarities between the two sequences cannot be taken advantage of.

**Procedure calls.** Consider two procedure calls \(f()\) and \(g()\) with bodies \(p\) and \(q\) respectively. Then:

\[
[f(), g()] = [p, q]
\]
Conditional branching. Consider two conditional statements if \( (c) \{ p \} \) else \( q \) and if \( (c) \{ p' \} \) else \( q' \) that agree on their conditional expressions. Then:

\[
\begin{align*}
\text{if} \ (c) \{ p \} \text{ else } q, \\
\text{if} \ (c) \{ p' \} \text{ else } q'
\end{align*}
\]

If the condition is evaluated equally in both branches, the composition should align the corresponding branches.

Iterative statements. Consider two while-loops

\[
\begin{align*}
\text{while} \ (c) \{ p \}, \\
\text{while} \ (c) \{ p' \}
\end{align*}
\]

that again agree on their conditional expressions. Then their composition is given by:

\[
\begin{align*}
\text{while} \ (c) \{ p \}, \\
\text{while} \ (c) \{ p' \}
\end{align*}
\]

It is possible that the loop in one component terminates, while the corresponding loop for the other component continues to run. Therefore, in the self-composition we need three loops, each representing a combination of possible evaluations of the pair of conditions.

These requirements are enough to show the comparison result to type system-based approaches in Section VI. However, we can improve upon our results by relying on a more elaborate requirement for conditional branching also met by the construction in [7, 24].

Conditional branching. Consider two conditional statements if \( (c) \{ p \} \) else \( q \) and if \( (c) \{ p' \} \) else \( q' \) that agree on their conditional expressions. Then:

\[
\begin{align*}
\text{if} \ (c) \{ p \} \text{ else } q, \\
\text{if} \ (c) \{ p' \} \text{ else } q'
\end{align*}
\]

Here, the composition of the two conditionals distinguishes four possible cases according to all possible combinations of values of the condition \( c \) when evaluated on the first and second component of the program state. This allows us to align similar behavior in the two branches and increase the precision of our analysis. In the following, we will assume that the self-composition operation satisfies this stronger requirement.

III. VARIABLE DEPENDENCIES BY PRECONDITION COMPUTATION

Assume that we are given a program variable \( x \). Our goal in this section is to determine a safe superset \( Y \) of program variables whose values at program start may influence the value of \( x \) at program exit. Formally, we are interested in a set of variables \( Y \subseteq G \), such that final states \( \sigma', \tau' \) with \( p : \sigma \rightsquigarrow \sigma' \) and \( p : \tau \rightsquigarrow \tau' \) agree on the values of \( x \) whenever the initial states \( \sigma, \tau \) before execution of \( p \) agree on all values of variables in \( Y \).

Our goal is to compute such a set \( Y \) by means of an abstract weakest precondition calculus on the self-composition of \( p \). Assertions \( \varphi \) are positive Boolean combinations of atomic assertions. An atomic assertion asserts that the two components of a pair of states agree on the value of a given program variable \( y \). For brevity, this statement is denoted by \( y \) itself. Accordingly, \( (\sigma, \tau) \models y \) iff \( \sigma(y) = \tau(y) \). As usual, we extend the satisfaction relation \( \models \) from atomic assertions to arbitrary positive Boolean combinations \( \varphi \) of atomic assertions.

We now introduce an abstract weakest precondition calculus \( \text{WP}' \) to compute for each assertion \( \varphi \), a precondition w.r.t. two
given programs $p$ and $p'$. This precondition of $\varphi$, as calculated by our calculus, is denoted by $\wp^# [p, p'] \varphi$.

Example 3. Consider a program $p$ with variables $x, y, z$, and assume that

$$\wp^# [p, p'] (x) = y \land z$$

In this case the values of $x$ after two executions of $p$ agree whenever the initial states before the execution have agreed on the values of program variables $y$ and $z$. □

The definition of $\wp^#$ is presented in Figure 3. For the moment, we consider programs without procedures only. Later in Section VII we will show how our methods can be extended to procedures as well. Consider a pair of identical assignments $x \leftarrow e$. In this case, the values of $x$ in the two components coincide whenever the values of all variables occurring in $e$ have coincided before the assignment. Therefore, the precondition is obtained from $\varphi$ by substituting the variable $x$ in $\varphi$ with the conjunction of the set $\text{vars}(e)$ of atomic assertions corresponding to the variables occurring in $e$. Consider a pair of any program fragment $t$ and $\text{skip}$. Then the values of a variable $x$ possibly modified by $t$ may differ. Accordingly, every $x \in \text{mod}(t)$ occurring in the post-condition $\varphi$ is replaced with $\texttt{fff}$. The definition of the set $\text{mod}(t)$ of possibly modified variables is presented in Figure 4.

Now consider the $\wp^#$ transformation of the composition of a pair of conditional statements $\text{if} \{c\} \{b_1\} \text{else} \{b_2\}$ and $\text{if} \{c\} \{b_1'\} \text{else} \{b_2'\}$. According to the composition operator, the composition of the two statements results in a case distinction on the respective outcomes of the condition $c$ for the two components of the state. This case distinction is reflected in the definition of $\wp^#$. The conjunction $\bigwedge \text{vars}(c)$ guarantees that the evaluation of $c$ returns the same result for both components. In this case, the first two conjuncts in the precondition provide sufficient conditions for $\varphi$ to hold. Otherwise, i.e., if $c$ may possibly evaluate to different values for the two components, then any composition of the alternatives provided by the two conditionals may occur. This is taken care of by the extra preconditions introduced in the second disjunct.

Example 4. Consider $p$ used in Example 1 with $[p, p]$ from Example 2. Then

$$\wp^# [p, p] x = \wp^# [y \leftarrow y + 1, y \leftarrow y + 1] (\wp^# [p', p'] [x])$$

where $p'$ abbreviates the conditional statement of the program, i.e., equals

$$\begin{align*}
\text{if} \ (\text{secret} = 0) \ {\{} \\
\quad x \leftarrow 0; \\
\quad y \leftarrow y + 1; \\
\} \ \text{else} \ {\{} \\
\quad y \leftarrow y + 1; \\
\}
\end{align*}$$

The formula $\wp^# [p', p'] (x)$ can be calculated as shown in Figure 5. As a result, we obtain that $\wp^# [p, p] (x) = \texttt{tt}$ and hence,

$$\begin{align*}
\wp^# [p, p] (x) &= \wp^# [y \leftarrow y + 1, y \leftarrow y + 1] (\texttt{tt}) \\
&= \texttt{tt}
\end{align*}$$

This means that on every possible run of $p$, $x$ will end up with the same value.

Finally, consider the $\wp^#$ transformation of the composition of a pair of iterative statements $t = \textbf{while} \{c\} \{p\}$ and $t' = \textbf{while} \{c\} \{p'\}$. In this case, the $\wp^#$ transformation is defined by means of the closure operator $(\_)^*$ applied to the precondition transformation corresponding to the bodies $p, p'$ of the loops. For an arbitrary monotonic transformation $T$, the operator $(\_)^*$ is defined by:

$$T^* \varphi = \bigwedge_{i \geq 0} T^i \varphi$$

Note that the reflexive and transitive closure operator $(\_)^*$ can equivalently be expressed as the greatest solution (w.r.t. the implication ordering) of the fixpoint equation:

$$X \varphi = \varphi \land T(X \varphi)$$

where $X$ is a monotonic transformation of positive Boolean combinations. This recursive definition corresponds to a tail-recursive representation of the loop. The case where all iterations of the two loops are executed in sync, is taken
The proof sketch. The proof consists of the following steps. First, we relate \( \wp^{\#} \) to proofs in a relational Hoare calculus for \( p \) and \( p' \) derived from the Hoare proof rules for singular programs. For each pair of program executions of \( p \) and \( p' \), transforming initial states \( \sigma, \tau \) into final states \( \sigma', \tau' \), respectively, the assertion of the theorem follows from the correctness of the Hoare proof rules for \( p \) and \( p' \). The soundness of \( \wp^{\#} \) does not depend on the requirements for the specific program composition operator, but only on the property (S) that the resulting self-composition encodes any possible pair of executions.

The proof system for relational Hoare logic is presented in Figure 7 and allows to infer properties of pairs of programs thus operating on pairs of states. The rules are essentially obtained from the original Hoare logic [36] and and follow the spirit of the proof method in [6].

Assertions \( A, B, C \), I mentioned in the proof rules may refer to values of variables in either of the two states. In order to provide an explicit notation for that in assertions, we refer to the value of a variable \( x \) in the first and second component of the state by \( x_1 \) and \( x_2 \), respectively. Likewise for an expression \( e \), we denote the expression \( e \) where every variable is replaced by its indexed version, with \( [e]_1 \) and \( [e]_2 \). The same convention also applies to program states \( \sigma, \tau \). Accordingly, an atomic assertion \( x \) in a Boolean combination \( \varphi \), which we use in our specification of the transformation \( \wp^{\#} \), is now considered as a shortcut for the assertion \( x_1 = x_2 \) — indicating that the values of \( x \) in both components are equal.

Moreover, we add the derived rule:

\[
\{ A \} p \mid p' \{ C \} \quad \{ B \} p \mid p' \{ C \}
\]

The rules ending in “Left” or “Right” are derived from the original Hoare calculus. Rules ending in “Align” can be derived by replacing the aligned statement \([st, st']\) by \([st, skip]; [skip, st']\) and using the “Left” and “Right” rules.

IV. CONJUNCTIVE REFORMULATION OF \( \wp^{\#} \)

The definition in Figure 3 requires the use of disjunctions for specifying the preconditions of conditional and iterative statements. In this section, we prove that conjunctions of atomic assertions are sufficient for computing \( \wp^{\#} \) for atomic postconditions. This result will allow us to improve significantly upon the algorithm given in Section III and arrive at a polynomial time algorithm. We introduce a new transformation \( \wp^{\#'} \) which refers exclusively to conjunctions. Its definition is shown in Figure 8.

Since all right-hand sides distribute over conjunctions, it suffices to specify the precondition transformation for atomic assertions only (instead of conjunctions of these). The precondition \( \wp^{\#'}[p, q]\varphi \) for a conjunction \( \varphi = \bigwedge_{y \in Y} y \) is then obtained by \( \bigwedge_{y \in Y} \wp^{\#'}[p, q]y \). In this definition, the operator \( (\bigwedge)\) is applied to a transformation of conjunctions \( \varphi \) and thus also returns a conjunction.

Subsequently, we prove that the two definitions, \( \wp^{\#} \) and \( \wp^{\#'} \), are actually equivalent. We state the following two lemmas.
For two programs \( a \) and \( b \), be program fragments, and \( \phi \) and \( \psi \) and \( \land \) and \( \lor \) if \( \psi \) and \( \text{skip} \) and \( \text{while} \), we proceed by induction on the structure of programs.

\[
\begin{array}{|c|c|}
\hline
\text{Iteration} & \text{Approximation for } \text{WP}^\# [t, t] x \\
\hline
0 & \text{tt} \\
1 & \text{tt} \land (x \land x \lor \ell \ell) = x \\
2 & x \land (x \land y \lor \ell \ell) = x \land y \\
3 & (x \land y) \land (y \land z \lor \ell \ell) = x \land y \land z \\
4 & (x \land y \land z) \land (y \land z \lor \ell \ell) = x \land y \land z \\
\hline
\end{array}
\]

Fig. 6. Approximations encountered in the fixpoint computation for Example 5.

\[
\begin{array}{|c|c|}
\hline
\text{Iteration} & \text{Approximation for } \text{WP}^\# [t, t] z \\
\hline
0 & \text{tt} \\
1 & \text{tt} \land (x \land z \lor \ell) = z \\
2 & z \land (z \lor \ell) = z \\
\hline
\end{array}
\]

Fig. 7. A proof system for relational Hoare logic.

**Lemma 1.** For two programs \( p, q \), if \( \text{WP}^\# [p, q] x \neq \ell \ell \), then

\[
\text{WP}^\# [p, q] x \equiv \text{WP}^\# [p, p] x \equiv \text{WP}^\# [q, q] x
\]

The proof is by induction on the structure of \( p \) and \( q \).

**Lemma 2.** Let \( p \) and \( q \) be program fragments, and \( \varphi \) a postcondition for the set of variables \( X \). If \( X \cap (\text{mod}(p) \cup \text{mod}(q)) = \emptyset \), then

\[
\text{WP}^\# [p, q] \varphi = \varphi
\]

The proof is again by induction on the structure of \( p \) and \( q \).

**Theorem 2.** Assume that \( p \) and \( q \) are program fragments without procedure calls. Then

\[
\text{WP}^\# [p, q] x \equiv \text{WP}^\# [p, q] x
\]

for every program variable \( x \).

*Proof.* We proceed by induction on the structure of programs. For the base cases \( \text{skip} \) and assignments, the definitions of \( \text{WP}^\# \) and \( \text{WP}^\#' \) are syntactically equal. Therefore, the statement of the theorem trivially holds. Since for concatenation, the definitions also agree, the statement there follows by inductive hypothesis. Therefore, it remains to consider the definitions for \( \text{if} \) and \( \text{while} \).

\[
\begin{array}{|c|c|}
\hline
\text{Iteration} & \text{Approximation for } \text{WP}^\# [t, t] z \\
\hline
0 & \text{tt} \\
1 & \text{tt} \land (x \land z \lor \ell) = z \\
2 & z \land (z \lor \ell) = z \\
\hline
\end{array}
\]

Fig. 7. A proof system for relational Hoare logic.

**Lemma 1.** For two programs \( p, q \), if \( \text{WP}^\# [p, q] x \neq \ell \ell \), then

\[
\text{WP}^\# [p, q] x \equiv \text{WP}^\# [p, p] x \equiv \text{WP}^\# [q, q] x
\]

The proof is by induction on the structure of \( p \) and \( q \).

**Lemma 2.** Let \( p \) and \( q \) be program fragments, and \( \varphi \) a postcondition for the set of variables \( X \). If \( X \cap (\text{mod}(p) \cup \text{mod}(q)) = \emptyset \), then

\[
\text{WP}^\# [p, q] \varphi = \varphi
\]

The proof is again by induction on the structure of \( p \) and \( q \).

**Theorem 2.** Assume that \( p \) and \( q \) are program fragments without procedure calls. Then

\[
\text{WP}^\# [p, q] x \equiv \text{WP}^\# [p, q] x
\]

for every program variable \( x \).

*Proof.* We proceed by induction on the structure of programs. For the base cases \( \text{skip} \) and assignments, the definitions of \( \text{WP}^\# \) and \( \text{WP}^\#' \) are syntactically equal. Therefore, the statement of the theorem trivially holds. Since for concatenation, the definitions also agree, the statement there follows by inductive hypothesis. Therefore, it remains to consider the definitions for \( \text{if} \) and \( \text{while} \).

If \( t, t' \) are conditional statements given by \( \text{if} (c) \{ p \} \text{ else } \{ q \} \) and \( \text{if} (c) \{ p' \} \text{ else } \{ q' \} \), respectively.
\[ \text{WP}''[y \leftarrow c, y \leftarrow e]x = \begin{cases} \text{if } x \neq y \\
\{ \text{vars}(c) \} \text{ otherwise} \end{cases} \]

\[ \text{WP}''[\text{skip}, st]x = \text{WP}''[st, \text{skip}]x = \begin{cases} \text{if } x \in \text{mod}(st) \\
\{ x \} \text{ otherwise} \end{cases} \]

\[ \text{WP}''[s_1 \ldots s_m, s'_1 \ldots s'_n]x = \begin{cases} \text{WP}''[t_1, t'_1 \ldots (\text{WP}''[t_k, t'_k]x) \ldots] \text{ where } [s_1 \ldots s_m, s'_1 \ldots s'_n] = [t_1, t'_1 \ldots [t_k, t'_k]\end{cases} \]

\[ \text{WP}''[\text{if } (c) \{ p \} \text{ else } q, \text{if } (c) \{ p' \} \text{ else } q']x = \begin{cases} \text{if one of } \text{WP}''[p, p']x, \text{WP}''[q, q']x, \\
\{ \text{WP}''[p, p']x \land \text{WP}''[q, q']x \} \text{ equals } x \in \text{mod}(p) \cup \text{mod}(p') \}
\text{otherwise} \end{cases} \]

\[ \text{WP}''[\text{while } (c) \{ p \}, \text{while } (c) \{ p' \}]x = \begin{cases} \text{if } x \notin \text{mod}(p) \cup \text{mod}(p') \}
\{ \text{WP}''[p, p']x \} (\text{vars}(c) \land x) \}
\text{otherwise} \end{cases} \]

Fig. 8. Conjunctive definition of \text{WP}''.

\textbf{Case 1.} One of \text{WP}''[p, p']x, \text{WP}''[q, q']x, \text{WP}''[p, q']x or \text{WP}''[q, p']x equals \text{ff}. By inductive hypothesis, we have

\[ \text{WP}''[p, p']x = \text{WP}''[q, q']x \]

and

\[ \text{WP}''[q, q']x = \text{WP}''[q, q']x \]

If either \text{WP}''[p, q']x or \text{WP}''[p', q']x equals \text{ff}, then

\[ \text{WP}''[t, t']x = \begin{cases} \text{WP}''[p, p']x \land \text{WP}''[q, q']x \land (\text{vars}(c) \lor \text{WP}''[p, q']x) \land (\text{WP}''[p', q']x \land (\text{vars}(c) \lor \text{ff})) \end{cases} \]

\[ \text{WP}''[p, p']x \land \text{WP}''[q, q']x \land (\text{vars}(c) \lor \text{ff}) \]

\[ \text{WP}''[p, p']x \land \text{WP}''[q, q']x \land (\text{vars}(c) \lor \text{ff}) \]

\[ \text{WP}''[t, t']x \]

Otherwise, \text{WP}''[t, t']x = \text{ff} = \text{WP}''[t, t']x.

\textbf{Case 2.} None of \text{WP}''[p, p']x, \text{WP}''[q, q']x, \text{WP}''[p, q']x and \text{WP}''[q, p']x equals \text{ff}. Then from Lemma 1, it follows that:

\[ \text{WP}''[p, p']x = \text{WP}''[p, p']x \]

\[ \text{WP}''[q, q']x = \text{WP}''[q, q']x \]

\[ \text{WP}''[p, p']x = \text{WP}''[q, q']x \]

\[ \text{WP}''[q, q']x \land \text{WP}''[p, p']x \land \text{WP}''[q, q']x \land \text{WP}''[p, p']x \land \text{WP}''[q, q']x \]

We conclude:

\[ \text{WP}''[t, t']x = \begin{cases} \text{WP}''[p, p']x \land \text{WP}''[q, q']x \land (\text{vars}(c) \lor \text{WP}''[p, q']x) \land (\text{WP}''[p', q']x \land (\text{vars}(c) \lor \text{WP}''[p', q']x) \land (\text{WP}''[p', q']x \land \text{WP}''[q, q']x) \land (\text{WP}''[p', q']x \land \text{WP}''[q, q']x) \land \text{WP}''[t, t']x \end{cases} \]

and equivalence follows.

\textbf{While.} \( t, t' \) are iterative statements given by \text{while } (c) \{ p \} and \text{while } (c) \{ p' \}, respectively. We distinguish two cases.

\textbf{Case 1.} \( x \notin \text{mod}(p) \cup \text{mod}(p') \). Then for the postcondition \( x \),

\[ \text{WP}''[t, t']x = x = \text{WP}''[t, t']x \]

by Lemma 2, and equivalence follows.

\textbf{Case 2.} \( x \in \text{mod}(p) \cup \text{mod}(p') \). Then

\[ \text{WP}''[t, t']x = \begin{cases} \text{WP}''[p, p']x \} \land \text{WP}''[q, q']x \land \text{WP}''[p, q']x \land \text{WP}''[q, p']x \land \text{WP}''[t, t']x \}
\text{otherwise} \end{cases} \]

holds by inductive hypothesis for \( p, p' \), and equivalence follows.

\[ \text{WP}''[t, t']x = \begin{cases} \text{WP}''[p, p']x \} \land \text{WP}''[q, q']x \land \text{WP}''[p, q']x \land \text{WP}''[q, p']x \land \text{WP}''[t, t']x \}
\text{otherwise} \end{cases} \]

By Theorem 2, the transformations as defined in Figures 3 and 8, respectively, are equivalent for postconditions \( x \in G \). In the sequel, we therefore no longer distinguish between the two transformations and denote the purely conjunctive transformation by \text{WP}'' as well.

\textbf{Theorem 3.} For every pair of program fragments \( t, t' \) and every program variable \( x \), \text{WP}''[t, t']x can be computed in polynomial time.

\[ \text{Proof.} \] First, we observe that \text{mod}(t) can be determined in polynomial time for all program fragments \( t \). Therefore, \text{WP}''[t, t']x can be determined in polynomial time whenever either \( t \) or \( t' \) equals \text{skip}. For arbitrary program fragments \( t, t' \), assume that we are given a corresponding alignment into a 2-program \( [t, t'] \). Such an alignment can be found in polynomial time and results in a 2-program of size at most quadratic in the the sizes of \( t, t' \) [7]. While maintaining a conjunction of atomic
from [14] where it serves as the motivation why flow-sensitivity matters for the analysis of information flow. We have that 
\[[p, p] = [x \leftarrow a, x \leftarrow a]; [x \leftarrow y, x \leftarrow y]\]
and accordingly,
\[\text{WP}^*[p, p][x = \text{WP}^*[x \leftarrow a, x \leftarrow a](\text{WP}^*[x \leftarrow y, x \leftarrow y](x))\]
\[= \text{WP}^*[x \leftarrow a, x \leftarrow a](y)\]
\[= y\]
Now assume that at program start a is secret, while x, y are public. Since \(\text{WP}^*[p, p][x = y\text{ and } y\text{ is public, the value of } x\text{ at program exit does not reveal any information about the secret. Thus, noninterference holds w.r.t. the variable } x.\]

B. Universal Information Flow

In general, an information flow analysis may distinguish more than just two security levels. Instead, an assignment of variables to security levels from some more complicated security lattice \(D\) is considered. Assume that \(\pi: G \rightarrow D\) assigns security levels to the program variables in the initial state. The program is considered as secure w.r.t. the variable \(x\) up to level \(d\), if the value of \(x\) at program exit only depends on input variables of security levels at most \(d\). As observed in [11], the analysis result for any specific lattice \(D\) can be retrieved from a single universal information flow analysis. This analysis uses the powerset of \(G\) (the set of program variables, ordered by subset inclusion) as information flow lattice with the initial assignment \(\pi\) with \(\pi(x) = \{x\}\). Universal information flow analysis thus determines for each variable \(x\) at program exit a safe superset \(Y\) of all variables whose values at program start may influence the value of \(x\) at program exit. The least security level for any other initial assignment \(\pi^*\) up to which the program is secure w.r.t. to the variable \(x\) then is obtained as \(\bigcup\{\pi^*(y)\mid y \in Y\}\).

In the case of the flow-sensitive type system of [11], the universal flow information is provided by means of a principal typing. By Theorem 1, computing \(\text{WP}^*[p, p][x\text{ for every program variable } x\text{ realizes another universal information flow analysis, as it also provides us with a safe superset of variable dependencies. As we will see in the next section, the sets provided by our analysis, though, are subsets of the sets provided by [11], sometimes even strict subsets. Still, by Theorem 3, our analysis runs in polynomial time.}

VI. COMPARISON WITH THE TYPE SYSTEM OF HUNT AND SANDS

A first flow-sensitive analysis of universal information flow is provided in [11]. This type-based analysis was shown to be equivalent to Hoare-like proof rules for information flow in [14]. Only later, it was shown that the given analysis can be realized in polynomial time [13]. As we have already seen in Example 1, our approach may improve upon the results of this
The definition of $[p]^T$ is recursive on the structure of $p$. We have:

**Lemma 3.** Consider a program fragment $p$ with $\vdash p : \Delta$. Then for every program variable $x$ we have:

$$\bigwedge (\Delta(x) \setminus \{pc\}) = [p]^T x$$

The proof is by induction on the structure of the program where the typing rules are in one-to-one correspondence to the definition of the transformation $[\_]^T$. In particular, the reflexive and transitive closure of the variable assignment $\Delta; \eta[p] \mapsto \text{vars}(c)$ is in one-to-one correspondence to the transformation $([p]^T)^*$ for the body $p$ of the loop.

As a next step, in Lemma 4 we show that the precondition computed by $[p]^T$ is stronger than the precondition computed by $\WP^p$ for the self-composition of $p$. Note that stronger is here meant in the logical sense, i.e., $[p]^T$ may include additional preconditions on equalities of variables which are revealed as unnecessary by $\WP^p$.

**Lemma 4.** For every program fragment $p$ and every program variable $x$,

$$[p]^T x \Rightarrow \WP^p [p, p]x$$

**Proof.** The proof again proceeds by induction on the structure of $p$. We prove according to the definition of $\WP^p$ presented in Figure 3.

**Inductive Hypothesis 1.** $[p]^T x \Rightarrow \WP^p [p, p]x$

**Skip.**

$$[\text{skip}]^T x = x = \WP^p [\text{skip}, \text{skip}]x$$

**Assign.**

$$[x \leftarrow c]^T x = \bigwedge \text{vars}(c) = \WP^p [x \leftarrow c, x \leftarrow c]x$$

$$[y \leftarrow c]^T x = x \quad \Rightarrow \WP^p [y \leftarrow c, y \leftarrow c]x$$

If. Assume that $x \not\in \text{mod}(p) \cup \text{mod}(q)$. Then

$$[\text{if } (c) \{p\} \text{ else } \{q\}]^T x = x = \WP^p [\text{if } (c) \{p\} \text{ else } \{q\}]x$$

Otherwise, we have:

$$[\text{if } (c) \{p\} \text{ else } \{q\}]^T x = \bigwedge \text{vars}(c) \land [p]^T x \land [q]^T x$$

$$\Rightarrow \bigwedge \text{vars}(c) \land \WP^p [p, p]x \land \WP^q [q, q]x$$

$$\Rightarrow \WP^p [p, p]x \land \WP^q [q, q]x$$

$$= \WP^p [\text{if } (c) \{p\} \text{ else } \{q\}]x, (c) \{p\} \text{ else } \{q\}]x$$

**While.** Assume that $x \not\in \text{mod}(p)$. Then

$$[\text{while } (c) \{p\}]^T x = x = \WP^p [\text{while } (c) \{p\}]x, (c) \{p\}]x$$
\[
\begin{align*}
\text{Skip} & \\ & \vdash \text{skip} : \eta \\
\text{Assign} & \\ & \vdash x := e : \eta[x \mapsto \{pc\} \cup \text{vars}(e)] \\
\text{Seq} & \\ & \vdash p : \Delta_2 \quad \vdash st : \Delta_1 \\
& \vdash st; p : \Delta_2; \Delta_1 \\
\text{If} & \\ & \vdash p : \Delta_1 \\
& \vdash q : \Delta_2 \\
& \Delta'_i = \Delta_i; \eta[pc \mapsto \text{vars}(e)] \quad (i = 1, 2) \\
& \vdash \text{if} (c) \{p\} \text{ else } \{q\} : (\Delta'_1 \cup \Delta'_2[pc \mapsto \{pc\}]) \\
\text{While} & \\ & \vdash \text{while} (c) \{p\} : (\Delta_f[pc \mapsto \{pc\}])
\end{align*}
\]

Fig. 9. Rules used to compute \( \vdash p : \Delta \).

\[
\begin{align*}
[\text{skip}]^T_x & = x \\
[x \leftarrow e]^T_z & = \begin{cases} \\
\text{vars}(e) & \text{if } x = z \\
\varnothing & \text{otherwise} \\
\end{cases} \\
[st; p']^T_x & = [st]^T(\{p'\}^T x) \\
[\text{if} (c) \{p\} \text{ else } \{q\}]^T_x & = \begin{cases} \\
\text{vars}(e) \land [p]^T x \land [q]^T x & \text{if } x \notin (\text{mod}(p) \cup \text{mod}(q)) \\
\varnothing & \text{otherwise} \\
\end{cases} \\
[\text{while} (c) \{p\}]^T_x & = \begin{cases} \\
\text{vars}(e) \land [\{p\}^T] (\text{vars}(e) \land x) & \text{if } x \notin \text{mod}(p) \\
\varnothing & \text{otherwise} \\
\end{cases}
\end{align*}
\]

Fig. 10. Backwards computation of \( \Delta(x) \) for \( \vdash p : \Delta \).

Otherwise, we have:
\[
\begin{align*}
[\text{while} (c) \{p\}]^T_x & = \{ [p]^T \}^* (\text{vars}(e) \land x) \\
& \Rightarrow (\text{WP}^\# [p, p])^* (\text{vars}(e) \land x) \\
& \Rightarrow (\text{WP}^\# [p, p])^* (\text{vars}(e) \land x) \lor \\
& \quad \varphi[x \notin (\text{mod}(p) \cup \text{mod}(p'))] \\
& \Rightarrow \text{WP}^\# [\text{while} (c) \{b\}], \text{while} (c) \{b\}]_x
\end{align*}
\]

Here, the implication follows since the operation \(( \lor )^*\) on monotonic transformations of conjunctions is monotonic.

\[
\square
\]

From Lemmas 3 and 4 we conclude that the information flow analysis by means of \(\text{WP}^\#\) is always at least as precise as the information flow analysis by means of the type system of Figure 9.

\[
\square
\]

VII. EXTENSION TO PROGRAMS WITH PROCEDURES

In this section, we extend the information flow analysis to programs consisting of multiple procedures which are possibly recursive. Assume that the procedures with identifiers \( f \) and \( g \) are defined by \( f()\{p\} \) and \( g()\{q\} \). First, let us extend the notion of the set of modified variables from Figure 4 to program fragments \( t \) possibly containing procedure calls. For that we add the following rule for procedure calls:
\[
\text{mod}(f()) = \text{mod}(p)
\]

Also, the definition of \( \text{mod} \) now has become recursive due to possibly recursive procedure calls. Here, we are interested in the least sets \( \text{mod}(t) \), where \( t \) is a program fragment, satisfying the definition.

Now we extend the definition of \( \text{WP}^\# \) as provided in Figure 8 with the following rules for procedure calls:
\[
\begin{align*}
\text{WP}^\# [f(), g()]_x & = \text{WP}^\# [p, q]_x \\
\text{WP}^\# [f(), \text{skip}]_x & = \text{WP}^\# [p, \text{skip}]_x \\
\text{WP}^\# [\text{skip}, f()]_x & = \text{WP}^\# [\text{skip}, p]_x
\end{align*}
\]

This definition is also recursive. The transformation which we are interested in is the greatest solution (w.r.t. the implication ordering) of the defining equations. The proof of the correctness of \( \text{WP}^\# \) as provided for Theorem 1 can be naturally extended to a correctness proof of \( \text{WP}^\# \) with procedure definitions. Also, Theorem 4 remains true in the presence of procedure calls using the extension of the type system from Figure 9 with the procedure typing rule in [13]. Likewise, the complexity result from Theorem 3 extends to the procedural case. We have:

Lemma 5. Assume that we are given two programs \( p, p' \) and a variable \( x \). Then all transformations \( \text{WP}^\# [f, g]_x \) for procedures \( f, g \) can jointly be computed in polynomial time.

For non-recursive procedures, this follows directly from the definition of \( \text{WP}^\# \) and Theorem 3. In order to deal with
possibly recursive procedure definitions, we perform a greatest fixpoint iteration. Since strictly decreasing chains of precondition transformers have length at most \( n^2 \) (\( n \) the number of program variables), a polynomial number of iterations suffices until the greatest fixpoint is reached. By Lemma 5, we can extend the result of Theorem 3 to programs with possibly recursive procedures:

**Theorem 5.** For every pair of program fragments \( t, t' \), and every program variable \( x \), \( \text{WP}^\# [t, t']x \) can be computed in polynomial time even in the presence of function calls.

The proof follows directly from Lemma 5.

**Example 8.** Consider the piece of code shown in Figure 11.

In the procedure `main`, procedure `hash` or procedure `constant_hash` are called, depending on the secret variable `secret_config`. Both procedures produce a result in the variable \( r \). For that, they access a global variable seed, together with further parameters (\( a \) or \( a, b \), respectively). Finally, the value of seed is incremented. We claim that the value of seed at program exit is independent of the values of `secret_base`, `secret_number`, `secret_config` at program start. This property cannot be proven by means of the interprocedural universal information flow of [13]. Even though the conditional depends on secret input and both branches change the seed variable, our method may align the calls `hash()` and `constant_hash()`. When aligning the bodies of the two procedures, our analysis realizes that

\[
\text{WP}^\# [\text{hash()}, \text{constant_hash()}](\text{seed}) = \text{seed}
\]

Since both branches affect the variable seed in the same way, we conclude that

\[
\text{WP}^\# [\text{main()}, \text{main()}](\text{seed}) = tt
\]

Therefore, the value of seed at program exit does not depend on secret variables at program start.

**VIII. RELATED WORK**

Based on type systems, a polynomial algorithm for analyzing universal information flow is presented in [11], [13]. An equi-expressive logic has already been presented in [14] without, however, discussing the complexity of the related algorithm. Our weakest precondition calculus based on self-composition of programs, improves on the precision of the type system in [13], while retaining polynomial complexity. An approach how to lift type systems to be able to handle conditional statements branching on secret-dependent variables is presented in [3]. However, no complexity for the approach is given.

The concept of information flow in programs has already been related to a relational Hoare logic and weakest precondition calculus, for example in [29], [2], [4]. Also [30] presents several proof systems based on Hoare logic to prove information flow properties of programs without, however, applying self-composition to programs. These systems do not provide explicit support for verifying programs where variables are manipulated inside branches having secret-dependent conditions.

Technically, our paper is based on a formalization of information flow as a hypersafety property [26] and relies on a self-composition of the program to align pairs of program executions. Self-composition of programs has been introduced in [12] and mentioned e.g. in [32]. The proof techniques there, however, generally do not take advantage of the similarities of different parts of the program. In [17] a type-directed method is presented to construct self-compositions of programs. In case of conditionals depending on the secret, the branching construct is doubled and composed sequentially, which explicitly rules out our optimizations. A similar approach can also be found in [27].

Proof techniques, on the other hand, which exploit similarities between different parts of the program include [6], [5], [1], [7], [34]. The authors of [6], [5], [1] present solutions relying on theorem proving. Similarly to our work, these approaches are based on a relational Hoare logic to prove properties of pairs of executions of the programs. In contrast to these, we have provided an abstract weakest precondition calculus to allow the automatic analysis of information flow.

In [34] it is shown how self-compositions can be applied to prove the differential privacy of probabilistic programs. In [7] an approach is presented, where relational abstract interpretation is applied to self-compositions in order to infer non-interference for tree-manipulating programs. These techniques, however, are not directly applicable to general security lattices. Our present approach on the other hand, overapproximates the set of variables in the initial state for

Fig. 11. Example code with function calls
each individual variable that can influence its final value. Since
the weakest precondition operator distributes over conjunction,
the conformance of the program to arbitrary information flow
policies can be verified based on the results of a single
analysis.

IX. CONCLUSION

We have presented a universal information flow analysis
based on an abstract weakest precondition computation on
self-compositions of programs. We compared this formulation
to the analysis of universal information flow based on type
systems as presented in [11], [13]. We showed that our algo-
rithm is always at least as precise as these type systems, and
sometimes may even gain precision over them. We showed that
our analysis still runs in polynomial time — even if procedures
are allowed. In this paper, for the sake of simplicity we
considered programs having global variables only. However,
our methods can be naturally enhanced to deal with local
variables as well.

As future work, we plan to further enhance the precision
of our analysis by combining it with additional relational
analyses of the program state. Also, we would like to extend
our approach to programs with objects and classes in order to
deal with real-world object-oriented languages such as JAVA
or C#.

REFERENCES

verification of information flow and access control policies,” ACM Trans.
ditional information flow,” in Proceedings of the 2007 ACM workshop
transformational typing and unification,” in Proceedings of Third
International Workshop on Formal Aspects in Security and Trust, FAST
45–60, revised version appeared in 2006 in Springer LNCS, [Online].
contract-based reasoning for verification and certification of information
flow properties of programs with arrays,” in Programming Languages
product programs for relational program verification,” in Logical Foun-
dations of Computer Science, International Symposium, (LICS), S. N.
Artemov and A. Nerode, Eds., 2013, pp. 29–43.
tion for the verification of 2-hypersafety properties,” in ACM Conference
on Computer and Communications Security, (CCS), A.-R. Sadeghi, V. D.
Conference on Computer and Communications Security. CCS'13, Berlin,
Germany, November 4-8, 2013, ACM, 2013.
ings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of
Programming Languages. (POPL), J. G. Morrisett and S. L. P. Jones,
Eds., 2006, pp. 79–90.
by self-composition,” in 17th IEEE Computer Security Foundations
typing via principal types,” in Programming Languages and Systems -
20th European Symposium on Programming, (ESOP), G. Barthe, Ed.,
in Static Analysis, 11th International Symposium, (SAS), R. Giacobazzi,
control and beyond,” in Proceedings of the 37th ACM SIGPLAN-SIGACT
Symposium on Principles of Programming Languages, (POPL), M. V.
in Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on
Principles of Programming Languages, (POPL), A. W. Appel and
object-sensitive information flow control based on program dependence
dependence graphs for representing programs,” in Conference Record
of the Fifteenth Annual ACM Symposium on Principles of Programming
secure flow analysis,” Journal of Computer Security, vol. 4, no. 2/3,
control,” in In Proc. 17th ACM Symp. on Operating System Principles
in 15th European Symposium on Research in Computer Security, (ES-
ORICS), D. Gritzalis, B. Preneel, and M. Thoehrindou, Eds., 2010,
pp. 116–133.
Aspects of Computer Science, (STACS), F.-J. Brandenburg, G. Vidal-
[24] H. Seidl and M. Kovács, “Interprocedural information flow analysis of
xml processors,” in Language and Automata Theory and Applications -
8th International Conference, (LATA), A. H. Dediu, C. Martín-Vide,
[27] D. A. Naumann, “From coupling relations to mated invariants for check-
ing information flow,” in ESORICS 2006, 11th European Symposium on
Research in Computer Security, ser. Lecture Notes in Computer Science,
D. Gollmann, J. Meier, and A. Sabelfeld, Eds., 4189, Springer,
ESORICS 2006, 11th European Symposium on Research in Computer
information flow in sequential programs,” in Computer Security -
ESORICS 94, Third European Symposium on Research in Computer
dx.doi.org/10.1007/3-540-58618-0_56
Symposium on Research in Computer Security, Brighton, UK, November


