2015 IEEE 28th Computer Security Foundations Symposium

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:*

$$
\begin{array}{l}
y \leftarrow 0;\\
\textbf{if}\,(secret = 0)\,\{\\
\quad x \leftarrow 0;\\
\quad y \leftarrow y + 1;\\
\}\,\textbf{else}\,\{\\
\quad x \leftarrow 0;\\
\}
\end{array}
$$

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

DOI 10.1109/CSF.2015.33

```
[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 `tt` *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:

$$
\begin{array}{lllll}
\text{(program)} & p & ::= & def_1 \dots def_k & \\
\text{(procedures)} & def & ::= & f()\{stmts\} & \\
\text{(statements)} & stmts & ::= & s_1 \dots s_n & \\
\text{(statement)} & s & ::= & x \leftarrow e; \mid f(); \mid \textbf{skip}; \mid & \\
 & & & \textbf{if}\,(c)\,\{stmts\}\,\textbf{else}\,\{stmts\} \mid & \\
 & & & \textbf{while}\,(c)\,\{stmts\} &
\end{array}
$$

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 σ, σ', the triple $p : \sigma \rightsquigarrow \sigma'$ denotes that the execution of program p on the initial state σ terminates with the final state σ'. 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 σ.

$$\frac{}{\textbf{skip} : \sigma \rightsquigarrow \sigma}$$

$$\frac{}{x \leftarrow e : \sigma \rightsquigarrow \sigma[x \mapsto \sigma[e]]}$$

$$\frac{st : \sigma \rightsquigarrow \sigma' \qquad p : \sigma' \rightsquigarrow \tau}{st; p : \sigma \rightsquigarrow \tau}$$

$$\frac{p_1 : \sigma \rightsquigarrow \tau}{\textbf{if}\,(c)\,\{p_1\}\,\textbf{else}\,\{p_2\} : \sigma \rightsquigarrow \tau}\ \text{if } \sigma \models c$$

$$\frac{p_2 : \sigma \rightsquigarrow \tau}{\textbf{if}\,(c)\,\{p_1\}\,\textbf{else}\,\{p_2\} : \sigma \rightsquigarrow \tau}\ \text{if } \sigma \models \neg c$$

$$\frac{p : \sigma \rightsquigarrow \sigma' \qquad \textbf{while}\,(c)\,\{p\} : \sigma' \rightsquigarrow \tau}{\textbf{while}\,(c)\,\{p\} : \sigma \rightsquigarrow \tau}\ \text{if } \sigma \models c$$

$$\frac{\texttt{tt}}{\textbf{while}\,(c)\,\{p\} : \sigma \rightsquigarrow \tau}\ \text{if } \sigma \models \neg c$$

$$\frac{p : \sigma \rightsquigarrow \tau}{f() : \sigma \rightsquigarrow \tau}\ \text{if } p \text{ is the body of } f$$

Fig. 2. Big-step Operational Semantics

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 **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 $\texttt{tt}$ iff both c_1 and c_2 evaluate to $\texttt{tt}$ 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 $pp : (\sigma_1, \sigma_2) \rightsquigarrow (\tau_1, \tau_2)$, where for the 2-program pp resulting from the *self-composition* $[p, p]$ we have:

(S) $pp : (\sigma, \tau) \rightsquigarrow (\sigma', \tau')$ iff $p : \sigma \rightsquigarrow \tau$ and $p : \tau \rightsquigarrow \tau'$.

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 $[s, \textbf{skip}]$ and $[\textbf{skip}, s]$ modify one state according to s and leave the other intact. Accordingly, we have:

$$[s_1 \ldots s_k, \textbf{skip}] = [s_1, \textbf{skip}] \ldots [s_k, \textbf{skip}]$$

$$\begin{aligned}[\textbf{if}\,(b)\,\{p_1\}\,\textbf{else}\,\{p_2\}, \textbf{skip}] = \ & \textbf{if}\,(b, \texttt{tt})\,\{ \\ & \quad [p_1, \textbf{skip}] \\ & \}\,\textbf{else}\,\{ \\ & \quad [p_2, \textbf{skip}] \\ & \}\end{aligned}$$

$$[\textbf{while}\,(b)\,\{p\}, \textbf{skip}] = \textbf{while}\,(b, \texttt{tt})\,\{[p, \textbf{skip}]\}$$

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

Here we assume that procedure f has body p. The rules for compositions $[\textbf{skip}, s]$ 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 $s'_1 \ldots s'_m$ and $t'_1 \ldots t'_r$ coincide — up to insertions of **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 **skip**, or one of the following properties holds:

- t, t' are syntactically identical assignments;
- both are procedure calls;
- both are **if**-statements with identical conditions; or
- both are **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 **skip** and then to concatenate the respective results as in:

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

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:

$$[f(), g()] = [p, q]$$

$$
\begin{array}{lcl}
\mathtt{WP}^{\#}[\![x \leftarrow e, x \leftarrow e]\!]\varphi & = & \varphi[\bigwedge vars(e)/x] \\
\mathtt{WP}^{\#}[\![\textbf{skip}, t]\!]\varphi = \mathtt{WP}^{\#}[\![t, \textbf{skip}]\!]\varphi & = & \varphi[\mathtt{ff}/x \mid x \in mod(t)] \\
\mathtt{WP}^{\#}[\![s_1 \dots s_m, s'_1 \dots s'_n]\!]\varphi & = & \mathtt{WP}^{\#}[\![t_1, t'_1]\!](\dots(\mathtt{WP}^{\#}[\![t_k, t'_k]\!]\varphi)\dots) \\
 & & \text{given that } [s_1 \dots s_m, s'_1 \dots s'_n] = [t_1, t'_1] \dots [t_k, t'_k] \\
\mathtt{WP}^{\#}\left[\!\!\left[\begin{array}{l} \textbf{if}\,(c)\,\{p\}\,\textbf{else}\,\{q\}, \\ \textbf{if}\,(c)\,\{p'\}\,\textbf{else}\,\{q'\} \end{array} \right]\!\!\right]\varphi & = & \mathtt{WP}^{\#}[\![p, p']\!]\varphi \wedge \mathtt{WP}^{\#}[\![q, q']\!]\varphi \wedge \\
 & & (\bigwedge vars(c) \vee (\mathtt{WP}^{\#}[\![p, q']\!]\varphi \wedge \mathtt{WP}^{\#}[\![q, p']\!]\varphi)) \\
\mathtt{WP}^{\#}\left[\!\!\left[\begin{array}{l} \textbf{while}\,(c)\,\{p\}, \\ \textbf{while}\,(c)\,\{p'\} \end{array} \right]\!\!\right]\varphi & = & (\mathtt{WP}^{\#}[\![p, p']\!])^{*}(\varphi \wedge \bigwedge vars(c)) \\
 & & \vee\, \varphi[\mathtt{ff}/(mod(p) \cup mod(p'))]
\end{array}
$$

Fig. 3. Definition of $\mathtt{WP}^{\#}$.

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

$$
\left[\begin{array}{l} \textbf{if}\,(c)\,\{p\}\,\textbf{else}\,\{q\}, \\ \textbf{if}\,(c)\,\{p'\}\,\textbf{else}\,\{q'\} \end{array} \right] = \begin{array}{l} \textbf{if}\,(b, b)\,\{ \\ \quad [p, p']\,; \\ \}\,\textbf{else if}\,(\neg b, \neg b)\,\{ \\ \quad [q, q']\,; \\ \}\,\textbf{else}\,\{ \\ \quad [\textbf{if}\,(c)\,\{p\}\,\textbf{else}\,\{q\}, \textbf{skip}]\,; \\ \quad [\textbf{skip}, \textbf{if}\,(c)\,\{p'\}\,\textbf{else}\,\{q'\}]\,; \\ \} \end{array}
$$

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

Iterative statements. Consider two **while**-loops **while** $(c)\,\{p\}$, **while** $(c)\,\{p'\}$ that again agree on their conditional expressions. Then their composition is given by:

$$
\left[\begin{array}{l} \textbf{while}\,(c)\,\{p\}, \\ \textbf{while}\,(c)\,\{p'\} \end{array} \right] = \begin{array}{l} \textbf{while}\,(c, c)\,\{[p, p']\}; \\ \textbf{while}\,(\neg c, c)\,\{[\textbf{skip}, p']\,;\}; \\ \textbf{while}\,(c, \neg c)\,\{[p, \textbf{skip}]\,;\}; \end{array}
$$

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:

$$
\left[\begin{array}{l} \textbf{if}\,(c)\,\{p\}\,\textbf{else}\,\{q\}, \\ \textbf{if}\,(c)\,\{p'\}\,\textbf{else}\,\{q'\} \end{array} \right] = \begin{array}{l} \textbf{if}\,(b, b)\,\{ \\ \quad [p, p']\,; \\ \}\,\textbf{else if}\,(\neg b, b)\,\{ \\ \quad [q, p']\,; \\ \}\,\textbf{else if}\,(b, \neg b)\,\{ \\ \quad [p, q']\,; \\ \}\,\textbf{else}\,\{ \\ \quad [q, q']\,; \\ \} \end{array}
$$

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 σ', τ' with $p : \sigma \rightsquigarrow \sigma'$ and $p : \tau \rightsquigarrow \tau'$ agree on the values of x whenever the initial states σ, τ 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 φ 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 φ of atomic assertions.

We now introduce an abstract weakest precondition calculus $\mathtt{WP}^{\#}$ to compute for each assertion φ, a precondition w.r.t. two

given programs p and p'. This precondition of φ, as calculated by our calculus, is denoted by $\mathtt{WP}^{\#}[\![p, p']\!]\varphi$.

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

$$\mathtt{WP}^{\#}[\![p, p]\!](x) = y \wedge z$$

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 $\mathtt{WP}^{\#}$ 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 φ by substituting the variable x in φ with the conjunction of the set $vars(e)$ of atomic assertions corresponding to the variables occurring in e. Consider a pair of any program fragment t and **skip**. Then the values of a variable x possibly modified by t may differ. Accordingly, every $x \in mod(t)$ occurring in the post-condition φ is replaced with $\mathtt{ff}$. The definition of the set $mod(t)$ of possibly modified

$$\begin{array}{lcl}
mod(\textbf{skip}) & = & \emptyset \\
mod(x \leftarrow e) & = & \{x\} \\
mod(t_1 \ldots t_k) & = & mod(t_1) \cup \ldots \cup mod(t_k) \\
mod(\textbf{if}\,(c)\,\{p\}\,\textbf{else}\,\{q\}) & = & mod(p) \cup mod(q) \\
mod(\textbf{while}\,(c)\,\{p\}) & = & mod(p)
\end{array}$$

Fig. 4. Definition of $mod(t)$.

variables is presented in Figure 4.

Now consider the $\mathtt{WP}^{\#}$ transformation of the composition of a pair of conditional statements $\textbf{if}\,(c)\,\{b_1\}\,\textbf{else}\,\{b_2\}$ and $\textbf{if}\,(c)\,\{b_1'\}\,\textbf{else}\,\{b_2'\}$. 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 $\mathtt{WP}^{\#}$. The conjunction $\bigwedge vars(c)$ 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 φ 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*

$$\mathtt{WP}^{\#}[\![p, p]\!]x = \mathtt{WP}^{\#}[\![y \leftarrow y+1, y \leftarrow y+1]\!](\mathtt{WP}^{\#}[\![p', p']\!]x)$$

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

if *(secret = 0)* {
    $x \leftarrow 0$;
    $y \leftarrow y + 1$;
} **else** {
    $y \leftarrow y + 1$;
}

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

$$\begin{array}{rcl}
\mathtt{WP}^{\#}[\![p, p]\!](x) & = & \mathtt{WP}^{\#}[\![y \leftarrow y+1, y \leftarrow y+1]\!](\mathtt{tt}) \\
& = & \mathtt{tt}
\end{array}$$

This means that on every possible run of p, x will end up

$$\begin{array}{rl}
& \mathtt{WP}^{\#}[\![p', p']\!](x) \\
= & \mathtt{WP}^{\#}[\![x \leftarrow 0, x \leftarrow 0]\!](\\
& \quad \mathtt{WP}^{\#}[\![y \leftarrow y+1, y \leftarrow y+1]\!](x) \\
&) \wedge \\
& \mathtt{WP}^{\#}[\![x \leftarrow 0, x \leftarrow 0]\!](x) \wedge (\\
& \quad \bigwedge vars(secret = 0) \vee \\
& \quad \mathtt{WP}^{\#}[\![x \leftarrow 0, x \leftarrow 0]\!](\\
& \quad\quad \mathtt{WP}^{\#}[\![y \leftarrow y+1, \textbf{skip}]\!](x) \\
& \quad) \wedge \\
& \quad \mathtt{WP}^{\#}[\![x \leftarrow 0, x \leftarrow 0]\!](\\
& \quad\quad \mathtt{WP}^{\#}[\![\textbf{skip}, y \leftarrow y+1]\!](x) \\
& \quad) \\
&) \\
= & \mathtt{tt} \wedge \\
& \mathtt{tt} \wedge (\\
& \quad secret \vee \mathtt{tt} \wedge \mathtt{tt} \\
&) \\
= & \mathtt{tt}
\end{array}$$

Fig. 5. $\mathtt{WP}^{\#}$ computation for a conditional

with the same value. □

Finally, consider the $\mathtt{WP}^{\#}$ 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 $\mathtt{WP}^{\#}$ 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:

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

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

care of by the first disjunct of the definition. It consists of $(\mathtt{WP}^{\#}[\![p,p]\!])^*$ applied to $\varphi \wedge \bigwedge vars(c)$. This means that the precondition transformer of the loop body is repeatedly applied to the postcondition as well as to $\bigwedge vars(c)$. The latter enforces that throughout the iteration, the values of all variables occurring in the condition will not differ in the two executions — implying that the two executions stay in sync. The second disjunct accounts for the case where the loop is not known to be executed in sync. In this case, all variables which are possibly modified may differ in their values. This is taken care of by substituting $\mathtt{ff}$ for every variable in φ which is possibly modified in one of the bodies p, p'. This set of variables is given by $mod(p) \cup mod(p')$.

The set of assertions possibly occurring during the evaluation of the $\mathtt{WP}^{\#}$ transformation is finite, since monotonic Boolean combinations of variables are closed under substitution, disjunction and conjunction. Moreover, since each case in the definition of $\mathtt{WP}^{\#}$ is monotonic (w.r.t. implication), the effect of the application of the operator $(\_)^*$ can be effectively computed as a *greatest* fixpoint, starting with the transformation that maps each postcondition to $\mathtt{tt}$.

Example 5. *Consider the composition $[t,t]$ where t is given by the loop*

while *($x > 0$)* {
  $x \leftarrow y - 1$;
  $y \leftarrow z - 1$;
}

Starting with the initial value $\mathtt{tt}$ for the precondition of x and z, we successively obtain the approximations to, for example, the preconditions $\mathtt{WP}^{\#}[\![t,t]\!]x$ and $\mathtt{WP}^{\#}[\![t,t]\!]z$ as shown in Figure 6. In this example, z is not modified in the loop, so as expected, the fixpoint iteration terminates quickly and find that z does not depend on any of the variables in the loop. For x, the fixpoint is reached after four iterations and depends on x, y, z. The dependency on y is from the assignment, which in turn is dependent on z. To ensure that the body of the loop is executed synchronously in both executions, all variables of the condition (in this case x) also have to be equal in both executions. □

Note that the definition of $\mathtt{WP}^{\#}$ makes explicit use of disjunctions. Since the precondition computation need not commute with disjunctions, the obvious implementation from the definition in Figure 3 may have to compute arbitrary conjunctions of disjunctions and therefore may take exponential time. Before we provide a reformulation which results in a more efficient algorithm, we convince ourselves that the $\mathtt{WP}^{\#}$ transformation is correct and thus allows to infer a sound approximation of the information flow in the program p.

Theorem 1. *Assume that $\mathtt{WP}^{\#}[\![p,p']\!]\varphi = \psi$ where $p : \sigma \rightsquigarrow \sigma'$ and $p' : \tau \rightsquigarrow \tau'$. Then $(\sigma', \tau') \models \varphi$ whenever $(\sigma, \tau) \models \psi$.*

Proof sketch. The proof consists of the following steps. First, we relate $\mathtt{WP}^{\#}$ 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 σ, τ into final states σ', τ', respectively, the assertion of the theorem follows from the correctness of the Hoare proof rules for p and p'. The soundness of $\mathtt{WP}^{\#}$ 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 σ, τ. Accordingly, an atomic assertion x in a Boolean combination φ, which we use in our specification of the transformation $\mathtt{WP}^{\#}$, 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:

$$\frac{\{A\}\ p \mid p'\ \{C\} \qquad \{B\}\ p \mid p'\ \{C\}}{\{A \vee B\}\ p \mid p'\ \{C\}}$$

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 $[st, \mathbf{skip}]; [\mathbf{skip}, st']$ and using the "Left" and "Right" rules. □

IV. Conjunctive Reformulation of $\mathtt{WP}^{\#}$

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 $\mathtt{WP}^{\#}$ 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 $\mathtt{WP}^{\#'}$ 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 $\mathtt{WP}^{\#'}[\![p,q]\!]\varphi$ for a conjunction $\varphi = \bigwedge_{y \in Y} y$ is then obtained by $\bigwedge_{y \in Y} \mathtt{WP}^{\#'}[\![p,q]\!]y$. In this definition, the operator $(\_)^*$ is applied to a transformation of conjunctions φ and thus also returns a conjunction.

Subsequently, we prove that the two definitions, $\mathtt{WP}^{\#}$ and $\mathtt{WP}^{\#'}$, are actually equivalent. We state the following two lemmas.

Iteration	Approximation for $\mathtt{WP}^{\#}[\![t,t]\!]x$	
0	$\mathtt{tt}$	
1	$\mathtt{tt} \wedge (x \wedge x \vee \mathtt{ff})$	$= x$
2	$x \wedge (x \wedge y \vee \mathtt{ff})$	$= x \wedge y$
3	$(x \wedge y) \wedge (y \wedge z \vee \mathtt{ff})$	$= x \wedge y \wedge z$
4	$(x \wedge y \wedge z) \wedge (y \wedge z \vee \mathtt{ff})$	$= x \wedge y \wedge z$

Iteration	Approximation for $\mathtt{WP}^{\#}[\![t,t]\!]z$	
0	$\mathtt{tt}$	
1	$\mathtt{tt} \wedge (x \wedge z \vee z)$	$= z$
2	$z \wedge (z \vee z)$	$= z$

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

$$\text{Skip}\ \frac{}{\{\varphi\}\ \textbf{skip} \mid \textbf{skip}\ \{\varphi\}}$$

$$\text{AssignLeft}\ \frac{}{\{\varphi[[e]_1/x_1]\}\ x \leftarrow e \mid \textbf{skip}\ \{\varphi\}}$$

$$\text{AssignRight}\ \frac{}{\{\varphi[[e]_2/x_2]\}\ \textbf{skip} \mid x \leftarrow e\ \{\varphi\}}$$

$$\text{AssignAlign}\ \frac{}{\{\varphi[[e]_1/x_1, [e]_2/x_2]\}\ x \leftarrow e \mid x \leftarrow e\ \{\varphi\}}$$

$$\text{ConcatLeft}\ \frac{\{\psi\}\ st \mid \textbf{skip}\ \{I\} \qquad \{I\}\ p \mid st'; p'\ \{\varphi\}}{\{\psi\}\ st; p \mid st'; p'\ \{\varphi\}}$$

$$\text{ConcatRight}\ \frac{\{\psi\}\ \textbf{skip} \mid st'\ \{I\} \qquad \{I\}\ st; p \mid p'\ \{\varphi\}}{\{\psi\}\ st; p \mid st'; p'\ \{\varphi\}}$$

$$\text{ConcatAlign}\ \frac{\{\psi\}\ st \mid st'\ \{I\} \qquad \{I\}\ p \mid p'\ \{\varphi\}}{\{\psi\}\ st; p \mid st'; p'\ \{\varphi\}}$$

$$\text{IfLeft}\ \frac{\{\psi \wedge [c]_1\}\ p \mid \textbf{skip}\ \{\varphi\} \qquad \{\psi \wedge \neg[c]_1\}\ q \mid \textbf{skip}\ \{\varphi\}}{\{\psi\}\ \textbf{if}\,(c)\,\{p\}\,\textbf{else}\,\{q\} \mid \textbf{skip}\ \{\varphi\}}$$

$$\text{IfRight}\ \frac{\{\psi \wedge [c]_2\}\ \textbf{skip} \mid p\ \{\varphi\} \qquad \{\psi \wedge \neg[c]_2\}\ \textbf{skip} \mid q\ \{\varphi\}}{\{\psi\}\ \textbf{skip} \mid \textbf{if}\,(c)\,\{p\}\,\textbf{else}\,\{q\}\ \{\varphi\}}$$

$$\text{IfAlign}\ \frac{\begin{array}{c}\{\psi \wedge ([c]_1 \wedge [c']_2)\}p \mid p'\{\varphi\},\\ \{\psi \wedge ([c]_1 \wedge \neg[c']_2)\}p \mid q'\{\varphi\},\\ \{\psi \wedge (\neg[c]_1 \wedge [c']_2)\}q \mid p'\{\varphi\},\\ \{\psi \wedge (\neg[c]_1 \wedge \neg[c']_2)\}\ q \mid q'\ \{\varphi\}\end{array}}{\{\psi\}\ \textbf{if}\,(c)\,\{p\}\,\textbf{else}\,\{q\} \mid \textbf{if}\,(c')\,\{p'\}\,\textbf{else}\,\{q'\}\ \{\varphi\}}$$

$$\text{WhileLeft}\ \frac{\begin{array}{c}\psi \Rightarrow I,\\ \{I \wedge [c]_1\}\ p \mid \textbf{skip}\ \{I\},\\ I \wedge \neg[c]_1 \Rightarrow \varphi\end{array}}{\{\psi\}\ \textbf{while}\,(c)\,\{p\} \mid \textbf{skip}\ \{\varphi\}}$$

$$\text{WhileRight}\ \frac{\begin{array}{c}\psi \Rightarrow I,\\ \{I \wedge [c]_2\}\ \textbf{skip} \mid p\ \{I\},\\ I \wedge \neg[c]_2 \Rightarrow \varphi\end{array}}{\{\psi\}\ \textbf{skip} \mid \textbf{while}\,(c)\,\{p\}\ \{\varphi\}}$$

$$\text{WhileAlign}\ \frac{\begin{array}{c}\psi \Rightarrow I,\\ \{I \wedge ([c]_1 \wedge [c]_2)\}p \mid p'\{I\},\\ I \wedge ([c]_1 \wedge \neg[c]_2) \Rightarrow J,\\ \{J \wedge ([c]_1 \wedge \neg[c]_2)\}p \mid \textbf{skip}\{J\},\\ I \wedge (\neg[c]_1 \wedge [c]_2) \Rightarrow J',\\ \{J' \wedge (\neg[c]_1 \wedge [c]_2)\}\textbf{skip} \mid p'\{J'\},\\ (J \vee J') \wedge \neg[c]_1 \wedge \neg[c]_2 \Rightarrow \varphi\end{array}}{\{\psi\}\ \textbf{while}\,(c)\,\{p\} \mid \textbf{while}\,(c')\,\{p'\}\ \{\varphi\}}$$

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

Lemma 1. *For two programs p,q, if* $\mathtt{WP}^{\#}[\![p,q]\!]x \neq \mathtt{ff}$*, then*

$$\mathtt{WP}^{\#}[\![p,q]\!]x \equiv \mathtt{WP}^{\#}[\![p,p]\!]x \equiv \mathtt{WP}^{\#}[\![q,q]\!]x$$

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

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

$$\mathtt{WP}^{\#'}[\![p,q]\!]\varphi = \varphi$$

□

The proof is again by induction on the structure of p and q. Now we prove our main theorem, which states the equivalence of $\mathtt{WP}^{\#}$ and $\mathtt{WP}^{\#'}$.

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

$$\mathtt{WP}^{\#}[\![p,q]\!]x \equiv \mathtt{WP}^{\#'}[\![p,q]\!]x$$

for every program variable x.

Proof. We proceed by induction on the structure of programs. For the base cases **skip** and assignments, the definitions of $\mathtt{WP}^{\#}$ and $\mathtt{WP}^{\#'}$ 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 **if** and **while**.

If. t, t' are conditional statements given by $\textbf{if}\,(c)\,\{p\}\,\textbf{else}\,\{q\}$ and $\textbf{if}\,(c)\,\{p'\}\,\textbf{else}\,\{q'\}$, respectively.

$$
\begin{array}{lcl}
\mathtt{WP}^{\#'}[\![y \leftarrow e, y \leftarrow e]\!]x & = & \begin{cases} x & \text{if } x \neq y \\ \bigwedge vars(e) & \text{otherwise} \end{cases} \\
\mathtt{WP}^{\#'}[\![\textbf{skip}, st]\!]x = \mathtt{WP}^{\#'}[\![st, \textbf{skip}]\!]x & = & \begin{cases} \mathtt{ff} & x \in mod(st) \\ x & \text{otherwise} \end{cases} \\
\mathtt{WP}^{\#'}[\![s_1 \ldots s_m, s'_1 \ldots s'_n]\!]x & = & \mathtt{WP}^{\#'}[\![t_1, t'_1]\!](\ldots(\mathtt{WP}^{\#'}[\![t_k, t'_k]\!]x)\ldots) \\
 & & \text{where } [s_1 \ldots s_m, s'_1 \ldots s'_n] = [t_1, t'_1]\ldots[t_k, t'_k] \\
\mathtt{WP}^{\#'}\left[\!\!\left[\begin{array}{l} \textbf{if}\,(\mathrm{c})\,\{\mathrm{p}\}\,\textbf{else}\,\{\mathrm{q}\}, \\ \textbf{if}\,(\mathrm{c})\,\{\mathrm{p'}\}\,\textbf{else}\,\{\mathrm{q'}\} \end{array} \right]\!\!\right] x & = & \begin{cases} \mathtt{WP}^{\#'}[\![p, p']\!]x \wedge \mathtt{WP}^{\#'}[\![q, q']\!]x \wedge \bigwedge vars(c) \\ \qquad \text{if one of } \mathtt{WP}^{\#'}[\![p, p']\!]x, \mathtt{WP}^{\#'}[\![q, q']\!]x, \\ \qquad \mathtt{WP}^{\#'}[\![p, q']\!]x, \mathtt{WP}^{\#'}[\![q, p']\!]x \text{ equals } \mathtt{ff} \\ \mathtt{WP}^{\#'}[\![p, q']\!]x \wedge \mathtt{WP}^{\#'}[\![q, p']\!]x \\ \qquad \text{otherwise} \end{cases} \\
\mathtt{WP}^{\#'}[\![\textbf{while}\,(c)\,\{p\}, \textbf{while}\,(c)\,\{p'\}]\!]x & = & \begin{cases} x & \text{if } x \notin mod(p) \cup mod(p') \\ \mathtt{WP}^{\#'}[\![p, p']\!]^*(\bigwedge vars(c) \wedge x) & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 8. Conjunctive definition of $\mathtt{WP}^{\#'}$.

Case 1. One of $\mathtt{WP}^{\#}[\![p, p']\!]x$, $\mathtt{WP}^{\#}[\![q, q']\!]x$, $\mathtt{WP}^{\#}[\![p, q']\!]x$ or $\mathtt{WP}^{\#}[\![q, p']\!]x$ equals $\mathtt{ff}$. By inductive hypothesis, we have

$$\mathtt{WP}^{\#}[\![p, p']\!]x \equiv \mathtt{WP}^{\#'}[\![p, p']\!]x$$

and

$$\mathtt{WP}^{\#}[\![q, q']\!]x \equiv \mathtt{WP}^{\#'}[\![q, q']\!]x$$

If either $\mathtt{WP}^{\#}[\![p, q']\!]x$ or $\mathtt{WP}^{\#}[\![q, p']\!]x \equiv \mathtt{ff}$, then

$$
\begin{array}{lll}
 & \mathtt{WP}^{\#}[\![t, t']\!]x & \\
= & \mathtt{WP}^{\#}[\![p, p']\!]x \wedge \mathtt{WP}^{\#}[\![q, q']\!]x \wedge & \\
 & \quad (\bigwedge vars(c) \vee \mathtt{WP}^{\#}[\![p, q']\!]x \wedge \mathtt{WP}^{\#}[\![q, p']\!]x) & \\
= & \mathtt{WP}^{\#}[\![p, p']\!]x \wedge \mathtt{WP}^{\#}[\![q, q']\!]x \wedge (\bigwedge vars(c) \vee \mathtt{ff}) & \\
\equiv & \mathtt{WP}^{\#'}[\![p, p']\!]x \wedge \mathtt{WP}^{\#'}[\![q, q']\!]x \wedge \bigwedge vars(c) & \\
= & \mathtt{WP}^{\#'}[\![t, t']\!]x &
\end{array}
$$

Otherwise, $\mathtt{WP}^{\#}[\![t, t']\!]x = \mathtt{ff} = \mathtt{WP}^{\#'}[\![t, t']\!]x$.

Case 2. None of $\mathtt{WP}^{\#}[\![p, p']\!]x$, $\mathtt{WP}^{\#}[\![q, q']\!]x$, $\mathtt{WP}^{\#}[\![p, q']\!]x$ and $\mathtt{WP}^{\#}[\![q, p']\!]x$ equals $\mathtt{ff}$. Then from Lemma 1, it follows that:

$$
\begin{array}{rcl}
\mathtt{WP}^{\#}[\![p, p']\!]x & \equiv & \mathtt{WP}^{\#}[\![p, p]\!]x \\
\mathtt{WP}^{\#}[\![p, q']\!]x & \equiv & \mathtt{WP}^{\#}[\![p, p]\!]x \\
\mathtt{WP}^{\#}[\![q, q']\!]x & \equiv & \mathtt{WP}^{\#}[\![q, q]\!]x \\
\mathtt{WP}^{\#}[\![q, p']\!]x & \equiv & \mathtt{WP}^{\#}[\![q, q]\!]x \\
\mathtt{WP}^{\#}[\![p, p']\!]x \wedge \mathtt{WP}^{\#}[\![q, q']\!]x & \equiv & \mathtt{WP}^{\#}[\![p, q']\!]x \wedge \mathtt{WP}^{\#}[\![q, p']\!]x
\end{array}
$$

We conclude:

$$
\begin{array}{lll}
 & \mathtt{WP}^{\#}[\![t, t']\!]x & \\
= & \mathtt{WP}^{\#}[\![p, p']\!]x \wedge \mathtt{WP}^{\#}[\![q, q']\!]x \wedge & \\
 & \quad (\bigwedge vars(c) \vee \mathtt{WP}^{\#}[\![p, q']\!]x \wedge \mathtt{WP}^{\#}[\![q, p']\!]x) & \\
\equiv & \mathtt{WP}^{\#}[\![p, q']\!]x \wedge \mathtt{WP}^{\#}[\![p', q]\!]x \wedge & \\
 & \quad (\bigwedge vars(c) \vee \mathtt{WP}^{\#}[\![p, q']\!]x \wedge \mathtt{WP}^{\#}[\![q, p']\!]x) & \\
\equiv & \mathtt{WP}^{\#}[\![p, q']\!]x \wedge \mathtt{WP}^{\#}[\![q, p']\!]x & \\
\equiv & \mathtt{WP}^{\#'}[\![p, q']\!]x \wedge \mathtt{WP}^{\#'}[\![q, p']\!]x & \\
= & \mathtt{WP}^{\#'}[\![t, t']\!]x &
\end{array}
$$

and equivalence follows.

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

Case 1. $x \notin mod(p) \cup mod(p')$. Then for the postcondition x,

$$\mathtt{WP}^{\#}[\![t, t']\!]x = x = \mathtt{WP}^{\#'}[\![t, t']\!]x$$

by Lemma 2, and equivalence follows.

Case 2. $x \in mod(p) \cup mod(p')$. Then

$$
\begin{array}{rcl}
\mathtt{WP}^{\#}[\![t, t']\!]x & = & (\mathtt{WP}^{\#}[\![p, p']\!])^*(\bigwedge vars(c) \wedge x) \vee \\
 & & \quad x[\mathtt{ff}/(mod(p) \cup mod(p'))] \\
 & = & (\mathtt{WP}^{\#}[\![p, p']\!])^*(\bigwedge vars(c) \wedge x) \vee \mathtt{ff} \\
 & \equiv & (\mathtt{WP}^{\#'}[\![p, p']\!])^*(\bigwedge vars(c) \wedge x) \\
 & = & \mathtt{WP}^{\#'}[\![t, t']\!]x
\end{array}
$$

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

□

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 $\mathtt{WP}^{\#}$ as well.

Theorem 3. *For every pair of program fragments t, t' and every program variable x, $\mathtt{WP}^{\#}[\![t, t']\!]x$ can be computed in polynomial time.* □

Proof. First, we observe that $mod(t)$ can be determined in polynomial time for all program fragments t. Therefore, $\mathtt{WP}^{\#}[\![t, t']\!]x$ can be determined in polynomial time whenever either t or t' equals **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

assertions, the algorithm proceeds by implicitly traversing the structure of the 2-program $[t, t']$. Let n denote the number of program variables. In order to tabulate the full transformer $\mathtt{WP}^{\#}[\![t, t']\!]$, we tabulate the corresponding transformer for each occurring 2-sub-program in a bottom-up way. Note that the composition of two tabulated transformers can be computed in polynomial time. By case distinction, we then convince ourselves that only polynomial extra effort is required to infer the transformer for the larger 2-subprogram from the transformers for its constituents. The only interesting case are aligned **while** loops. In this case, the transitive closure of the transformer for the aligned loop bodies must be determined — which is again is possible in polynomial time. □

V. Using $\mathtt{WP}^{\#}$ to capture Universal Information Flow

Let us briefly explain how our analysis is related to the notions of noninterference and (universal) information flow. In the following, we assume that the program operates on possibly classified values which are stored in the variables of the program. Each variable x is assigned a *security level* indicating the secrecy of the value of x at program start. The set D of all security levels is assumed to form a complete lattice (ordered by $\sqsubseteq$). For each observation of the program behavior possibly made by a principal, we may ask whether or not this observation reveals information beyond a given security level. The only observations we consider here, are the values of program variables at program exit.

A. Noninterference

Noninterference [25] is the instance of the given problem where only two security levels are considered, namely H (high/secret) and L (low/public). Noninterference holds for a program p w.r.t. a given variable x if the value of x at program exit only depends on the initial values of variables marked as L.

Assume that we have proven for program p that $\mathtt{WP}^{\#}[\![p, p]\!]x = \varphi$. According to Theorem 1, this means that two terminating executions of p result in the same value for x given the initial states of the two executions satisfy the conjunction $\varphi = \bigwedge Y$ for some set $Y \subseteq G$. Now consider two initial variable assignments σ, σ'. By the correctness of $\mathtt{WP}^{\#}$, we deduce that x holds at program exit whenever σ, σ' satisfy the assertion φ, or equivalently, $\sigma(y) = \sigma'(y)$ for all $y \in Y$, while the values for the remaining variables may be chosen arbitrarily. Therefore if all variables in Y are marked as L, the values of variables marked with H are irrelevant for the value of x at program exit, and noninterference follows. If on the other hand, one of the variables in Y is marked as H, we cannot exclude that the final value of x may depend on secret values at program start. In this sense, our analysis allows to prove noninterference w.r.t. a given variable x at program exit.

Example 6. *As an example, consider the program p given by:*

$$x \leftarrow a;$$
$$x \leftarrow y;$$

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,

$$\begin{aligned}\mathtt{WP}^{\#}[\![p, p]\!]x =& \mathtt{WP}^{\#}[\![x \leftarrow a, x \leftarrow a]\!](\\ & \mathtt{WP}^{\#}[\![x \leftarrow y, x \leftarrow y]\!](x) \\ &) \\ =& \mathtt{WP}^{\#}[\![x \leftarrow a, x \leftarrow a]\!](y) \\ =& y\end{aligned}$$

Now assume that at program start a is secret, while x, y are public. Since $\mathtt{WP}^{\#}[\![p, p]\!]x = y$ and y is public, the value of x 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 $\pi : G \to D$ 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 π 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 π' up to which the program is secure w.r.t. to the variable x then is obtained as $\bigsqcup\{\pi'(y) \mid y \in Y\}$.

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 $\mathtt{WP}^{\#}[\![p, p]\!]x$ for every program variable x 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

analysis even on very small examples. Here, we show that our analysis by means of $\mathtt{WP}^{\#}$ is always at least as precise as the analysis in [13].

The typing rules from [13] that can be used to infer the principal typing (i.e., the analysis result for program exit) are shown in Figure 9. The semantics is given in the form of judgments $\vdash p : \Delta$ where Δ maps variables to elements of $2^{G \cup \{\mathbf{pc}\}}$. It is a forward-semantic computing Δ, the principal typing of p, where a fresh variable **pc** represents the program counter. Furthermore, $\Delta; \Delta'$ means relational composition of typings, i.e. $(x \in \Delta; \Delta'(y))$ iff $\exists z. x \in \Delta(z) \wedge z \in \Delta'(y)$ and η denotes the identity mapping, i.e. $\eta(x) = \{x\}$. Whenever a variable is assigned, its new value depends on the variables on the right-hand-side and also on the control flow to that particular assignment. To be able to track the dependencies generated by the control flow, the rules for conditionals and loops compose the typing computed for their respective bodies with $\eta[\mathbf{pc} \mathrel{+}= vars(c)]$. Here **pc** is mapped to the variables of the branching expression. This way, the resulting typing contains the variables occurring in the condition for all variables assigned inside the bodies.
Here, we prove:

Theorem 4. *For every program fragment p without procedure calls where we have $\vdash p : \Delta$, and for every variable x the following holds:*

1) $\mathbf{pc} \in \Delta(x)$ *iff* $x \in mod(p)$*;*
2) *The conjunction* $\bigwedge(\Delta(x) \setminus \{\mathbf{pc}\})$ *implies* $\mathtt{WP}^{\#}[\![p, p]\!]x$.

As the results provided by the type system in general are less precise than the results computed by means of $\mathtt{WP}^{\#}$, the implication in the second statement cannot be replaced by an equivalence.

Example 7. *Consider the program used in Examples 1 and 4 with $H = \{secret, y\}$ and $L = \{x\}$. The type-system would return a typing $\vdash p : \Delta$ where*

$$\Delta(x) = \{secret, \mathbf{pc}\}$$

This implies that x is possibly modified by p (i.e., $x \in mod(p)$) and, moreover, that its final value may depend on the initial value of $secret$. On the other hand, as established in Example 4, $\mathtt{WP}^{\#}[\![p, p]\!]x = \mathtt{tt}$. Accordingly, x is proven to be independent *of all variables at program start.* □

Intuitively, more precision can be gained by $\mathtt{WP}^{\#}$ at conditionals where both branches modify a variable in the same way. As shown in Section VII, this advantage also holds in the presence of function calls.

Proof. We only prove the second assertion of Theorem 4. To be able to relate the semantics given in [13] with the semantics of $\mathtt{WP}^{\#}$ given in Section III, we reformulate the type system $\vdash p : \Delta$ as a backward operator $[\![p]\!]^{\mathsf{T}}$ as shown in Figure 10. This allows to omit an explicit treatment of the variable **pc**. Instead, the sets of possibly modified variables as defined in Figure 4, are referred to.

The definition of $[\![p]\!]^{\mathsf{T}}$ 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:*

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

□

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 $[\![\_]\!]^{\mathsf{T}}$. In particular, the reflexive and transitive closure of the variable assignment $\Delta; \eta[\mathbf{pc} \mathrel{+}= vars(c)]$ is in one-to-one correspondence to the transformation $([\![p]\!]^{\mathsf{T}})^*$ for the body p of the loop.

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

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

$$[\![p]\!]^{\mathsf{T}} x \Rightarrow \mathtt{WP}^{\#}[\![p, p]\!]x$$

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

Inductive Hypothesis 1. $[\![p]\!]^{\mathsf{T}} x \Rightarrow \mathtt{WP}^{\#}[\![p, p]\!]x$

Skip.

$$[\![\mathbf{skip}]\!]^{\mathsf{T}} x = x = \mathtt{WP}^{\#}[\![\mathbf{skip}, \mathbf{skip}]\!]x$$

Assign.

$$\begin{array}{lcll} [\![x \leftarrow e]\!]^{\mathsf{T}} x & = & \bigwedge vars(e) & = & \mathtt{WP}^{\#}[\![x \leftarrow e, x \leftarrow e]\!]x \\ [\![y \leftarrow e]\!]^{\mathsf{T}} x & = & x & = & \mathtt{WP}^{\#}[\![y \leftarrow e, y \leftarrow e]\!]x \\ & & & & \text{if } x \neq y \end{array}$$

If. Assume that $x \notin mod(p) \cup mod(q)$. Then

$$[\![\mathbf{if}\,(c)\,\{p\}\,\mathbf{else}\,\{q\}]\!]^{\mathsf{T}} x = x = \mathtt{WP}^{\#}[\![\mathbf{if}\,(c)\,\{p\}\,\mathbf{else}\,\{q\}, \mathbf{if}\,(c)\,\{p\}\,\mathbf{else}\,\{q\}]\!]x$$

Otherwise, we have:

$$\begin{array}{ll} & [\![\mathbf{if}\,(c)\,\{p\}\,\mathbf{else}\,\{q\}]\!]^{\mathsf{T}} x \\ = & \bigwedge vars(c) \wedge [\![p]\!]^{\mathsf{T}} x \wedge [\![q]\!]^{\mathsf{T}} x \\ \Rightarrow & \bigwedge vars(c) \wedge \mathtt{WP}^{\#}[\![p, p]\!]x \wedge \mathtt{WP}^{\#}[\![q, q]\!]x \\ \Rightarrow & \mathtt{WP}^{\#}[\![p, p]\!]x \wedge \mathtt{WP}^{\#}[\![q, q]\!]x \wedge \\ & (\bigwedge vars(c) \vee (\mathtt{WP}^{\#}[\![p, q]\!]x \wedge \mathtt{WP}^{\#}[\![q, p]\!]x) \\ = & \mathtt{WP}^{\#}[\![\mathbf{if}\,(c)\,\{p\}\,\mathbf{else}\,\{q\}, \mathbf{if}\,(c)\,\{p\}\,\mathbf{else}\,\{q\}]\!]x \end{array}$$

While. Assume that $x \notin mod(p)$. Then

$$[\![\mathbf{while}\,(c)\,\{p\}]\!]^{\mathsf{T}} x = x = \mathtt{WP}^{\#}[\![\mathbf{while}\,(c)\,\{p\}, \mathbf{while}\,(c)\,\{p\}]\!]x$$

$$\text{Skip}\ \frac{}{\vdash \mathbf{skip} : \eta}$$

$$\text{Assign}\ \frac{}{\vdash x := e : \eta[x \mapsto \{\mathbf{pc}\} \cup vars(e)]}$$

$$\text{Seq}\ \frac{\vdash p : \Delta_2 \qquad \vdash st : \Delta_1}{\vdash st; p : \Delta_2; \Delta_1}$$

$$\text{If}\ \frac{\vdash p : \Delta_1 \qquad \vdash q : \Delta_2 \qquad \Delta_i' = \Delta_i; \eta[\mathbf{pc}\ \mathbf{+\!\!=}\ vars(e)] \qquad (i = 1, 2)}{\vdash \mathbf{if}\,(\mathrm{c})\,\{\mathrm{p}\}\,\mathbf{else}\,\{\mathrm{q}\} : (\Delta_1' \sqcup \Delta_2'[\mathbf{pc} \mapsto \{\mathbf{pc}\}])}$$

$$\text{While}\ \frac{\vdash p : \Delta \qquad \Delta_f = (\Delta; \eta[\mathbf{pc}\ \mathbf{+\!\!=}\ vars(c)])^*}{\vdash \mathbf{while}\,(\mathrm{c})\,\{\mathrm{p}\} : (\Delta_f[\mathbf{pc} \mapsto \{\mathbf{pc}\}])}$$

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

$$\begin{aligned}
[\![\mathbf{skip}]\!]^{\mathsf{T}} x &= x \\
[\![x \leftarrow e]\!]^{\mathsf{T}} z &= \begin{cases} \bigwedge vars(e) & \text{if } x = z \\ z & \text{otherwise} \end{cases} \\
[\![st; p']\!]^{\mathsf{T}} x &= [\![st]\!]^{\mathsf{T}}([\![p']\!]^{\mathsf{T}} x) \\
[\![\mathbf{if}\,(c)\,\{p\}\,\mathbf{else}\,\{q\}]\!]^{\mathsf{T}} x &= \begin{cases} x & \text{if } x \notin (mod(p) \cup mod(q)) \\ \bigwedge vars(c) \wedge [\![p]\!]^{\mathsf{T}} x \wedge [\![q]\!]^{\mathsf{T}} x & \text{otherwise} \end{cases} \\
[\![\mathbf{while}\,(c)\,\{p\}]\!]^{\mathsf{T}} x &= \begin{cases} x & \text{if } x \notin mod(p) \\ ([\![p]\!]^{\mathsf{T}})^*(vars(c) \wedge x) & \text{otherwise} \end{cases}
\end{aligned}$$

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

Otherwise, we have:

$$\begin{array}{lll}
 & [\![\mathbf{while}\,(c)\,\{p\}]\!]^{\mathsf{T}} x \\
= & ([\![p]\!]^{\mathsf{T}})^*(\bigwedge vars(c) \wedge x) \\
\Rightarrow & (\mathtt{WP}^{\#}[\![p, p]\!])^*(\bigwedge vars(c) \wedge x) \\
\Rightarrow & (\mathtt{WP}^{\#}[\![p, p]\!])^*(\bigwedge vars(c) \wedge x) \vee \\
 & \qquad \varphi[\mathtt{ff}/(mod(p) \cup mod(p'))] \\
= & \mathtt{WP}^{\#}[\![\mathbf{while}\,(c)\,\{b\}, \mathbf{while}\,(c)\,\{b\}]\!] x
\end{array}$$

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

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

□

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:

$$mod(f()) \quad = \quad mod(p)$$

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

Now we extend the definition of $\mathtt{WP}^{\#}$ as provided in Figure 8 with the following rules for procedure calls:

$$\begin{array}{lcl}
\mathtt{WP}^{\#}[\![f(), g()]\!] x & = & \mathtt{WP}^{\#}[\![p, q]\!] x \\
\mathtt{WP}^{\#}[\![f(), \mathbf{skip}]\!] x & = & \mathtt{WP}^{\#}[\![p, \mathbf{skip}]\!] x \\
\mathtt{WP}^{\#}[\![\mathbf{skip}, f()]\!] x & = & \mathtt{WP}^{\#}[\![\mathbf{skip}, p]\!] x
\end{array}$$

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 $\mathtt{WP}^{\#}$ as provided for Theorem 1 can be naturally extended to a correctness proof of $\mathtt{WP}^{\#}$ 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 $\mathtt{WP}^{\#}[\![f, g]\!] x$ for procedures f, g can jointly be computed in polynomial time.* □

For non-recursive procedures, this follows directly from the definition of $\mathtt{WP}^{\#}$ and Theorem 3. In order to deal with

```
main() {
    seed ← 1;
    a ← secret_base;
    if (secret_config) {
        b ← secret_number;
        hash();
    } else {
        constant_hash();
    }
}

hash() {
    // Complicated Hash computation
    // based on the seed, a and b
    r ← seed * a * b;
    seed ← 1 + seed;
}
constant_hash() {
    // Complicated Hash computation
    // based only on the seed and a
    r ← seed * a * 42;
    seed ← 1 + seed;
}
```

Fig. 11. Example code with function calls

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, $\mathtt{WP}^{\#}[\![t, t']\!]x$ 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

$$\mathtt{WP}^{\#}[\![hash(), constant\_hash()]\!](seed) = seed$$

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

$$\mathtt{WP}^{\#}[\![main(), main()]\!](seed) = \mathtt{tt}$$

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

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 algorithm 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

[1] A. Nanevski, A. Banerjee, and D. Garg, "Dependent type theory for verification of information flow and access control policies," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 2, p. 6, 2013.

[2] T. Amtoft and A. Banerjee, "Verification condition generation for conditional information flow," in *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*. ACM, 2007, pp. 2–11.

[3] B. Köpf and H. Mantel, "Eliminating implicit information leaks by transformational typing and unification," in *Proceedings of Third International Workshop on Formal Aspects in Security and Trust, FAST 2005*. Newcastle, UK: University of Newcastle, July 18-19th 2005, pp. 45–60, revised version appeared in 2006 in Springer LNCS. [Online]. Available: http://www.iit.cnr.it/FAST2005/Unico.htm

[4] T. Amtoft, J. Hatcliff, and E. Rodríguez, "Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays," in *Programming Languages and Systems*. Springer, 2010, pp. 43–63.

[5] G. Barthe, J. M. Crespo, and C. Kunz, "Beyond 2-safety: Asymmetric product programs for relational program verification," in *Logical Foundations of Computer Science, International Symposium, (LFCS)*, S. N. Artëmov and A. Nerode, Eds., 2013, pp. 29–43.

[6] ——, "Relational verification using product programs," in *FM*, 2011, pp. 200–214.

[7] M. Kovács, H. Seidl, and B. Finkbeiner, "Relational abstract interpretation for the verification of 2-hypersafety properties," in *ACM Conference on Computer and Communications Security, (CCS)*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 211–222.

[8] A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds., *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 2013.

[9] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[10] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[11] S. Hunt and D. Sands, "On flow-sensitive security types," in *Proceedings 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.

[12] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)*, 2004, pp. 100–114.

[13] S. Hunt and D. Sands, "From exponential to polynomial-time security typing via principal types," in *Programming Languages and Systems - 20th European Symposium on Programming, (ESOP)*, G. Barthe, Ed., 2011, pp. 297–316.

[14] T. Amtoft and A. Banerjee, "Information flow analysis in logical form," in *Static Analysis, 11th International Symposium, (SAS)*, R. Giacobazzi, Ed. Springer, 2004, pp. 100–115.

[15] N. Broberg and D. Sands, "Paralocks: role-based information flow control and beyond," in *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL)*, M. V. Hermenegildo and J. Palsberg, Eds., 2010, pp. 431–444.

[16] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL)*, A. W. Appel and A. Aiken, Eds., 1999, pp. 228–241.

[17] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *SAS*, 2005, pp. 352–367.

[18] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *Int. J. Inf. Sec.*, vol. 8, no. 6, pp. 399–422, 2009.

[19] S. Horwitz, J. Prins, and T. W. Reps, "On the adequacy of program dependence graphs for representing programs," in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, (POPL)*, J. Ferrante and P. Mager, Eds., 1988, pp. 146–157.

[20] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996.

[21] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *In Proc. 17th ACM Symp. on Operating System Principles (SOSP)*. ACM, 1997, pp. 129–142.

[22] H. Mantel and H. Sudbrock, "Flexible scheduler-independent security," in *15th European Symposium on Research in Computer Security, (ESORICS)*, D. Gritzalis, B. Preneel, and M. Theoharidou, Eds., 2010, pp. 116–133.

[23] G. Kahn, "Natural semantics," in *4th Annual Symposium on Theoretical Aspects of Computer Science, (STACS)*, F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds., 1987, pp. 22–39.

[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, J. L. Sierra-Rodríguez, and B. Truthe, Eds., 2014, pp. 34–61.

[25] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[26] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[27] D. A. Naumann, "From coupling relations to mated invariants for checking 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., vol. 4189. Springer, 2006, pp. 279–296.

[28] D. Gollmann, J. Meier, and A. Sabelfeld, Eds., *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4189. Springer, 2006.

[29] R. Joshi and K. R. M. Leino, "A semantic approach to secure information flow," *Sci. Comput. Program.*, vol. 37, no. 1-3, pp. 113–138, 2000.

[30] J. Banâtre, C. Bryce, and D. L. Métayer, "Compile-time detection of information flow in sequential programs," in *Computer Security - ESORICS 94, Third European Symposium on Research in Computer Security, Brighton, UK, November 7-9, 1994, Proceedings*, ser. Lecture Notes in Computer Science, D. Gollmann, Ed., vol. 875. Springer, 1994, pp. 55–73. [Online]. Available: http://dx.doi.org/10.1007/3-540-58618-0_56

[31] D. Gollmann, Ed., *Computer Security - ESORICS 94, Third European Symposium on Research in Computer Security, Brighton, UK, November 7-9, 1994, Proceedings*, ser. Lecture Notes in Computer Science, vol. 875. Springer, 1994.

[32] Á. Darvas, R. Hähnle, and D. Sands, "A theorem proving approach to analysis of secure information flow," in *SPC*, 2005, pp. 193–209.

[33] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[34] G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, C. Kunz, and P. Strub, "Proving differential privacy in hoare logic," in *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE, 2014, pp. 411–424. [Online]. Available: http://dx.doi.org/10.1109/CSF.2014.36

[35] *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE, 2014. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6954678

[36] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[37] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, October 2009, supersedes ISSSE and ISoLA 2006. [Online]. Available: http://pp.info.uni-karlsruhe.de/uploads/publikationen/hammer09ijis.pdf:PDF

[38] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 1982, pp. 11–20. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SP.1982.10014

[39] *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 1982. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6234453