Type Systems Equivalent to Data-Flow Analyses of Imperative Languages (Extended Abstract)

Peeter Laud[1] and Tarmo Uustalu[2] and Varmo Vene[1]

[1]Institute of Computer Science, University of Tartu
[2]Institute of Cybernetics, Tallinn

Abstract

We show that a large class of data-flow analyses for imperative languages are describable as type systems in the following technical sense: possible results of an analysis can be described in a language of types so that a program checks with a type if and only if this type is a supertype of a result of applying the analysis. Type-checking is easy with the help of a certificate that records the "heureka"-bits of a derivation of the given type. Certificate-assisted type-checking amounts to a form of lightweight analysis à la Rose. For secure information flow, we obtain a type system that is considerably more precise than that of Volpano et al., but not more sophisticated. Needless to say, our type systems are compositional.

1 Introduction

Both program analyses and type systems can be used to infer and state properties of programs. Clearly, program analyses are more algorithmic by their nature while type systems are more declarative. In particular, although program analyses are convenient tools for inferring properties of programs, type systems have advantages over them, when it comes to communicating the properties established.

Given these considerations, it is most natural to wish to relate program analyses and type systems. This idea has been around for quite some time. In particular, Cousot has shown that both are instances of abstract interpretation [4] and there are several works describing how to derive type systems from program analysis definitions [11, 7]. But the project has not been carried out really satisfactorily for imperative languages. The main obstacle seems to have been a confusion over what a type system for an imperative language should look like.

In this paper, we tackle exactly the problem of deriving type systems from analyses for imperative languages (more specifically, data-flow analyses) and solve it based on a simple philosophical idea of what types are all about. Our position is that a type must always be an abstraction of the denotation of a program, be it then an abstraction of its denotation wrt. the standard semantics of language or some non-standard semantics. Since imperative programs denote transformations of (standard or non-standard) states—a state before a run of the program is sent to the state after it—, types for imperative programs must be *implications* between abstractions of (standard or non-standard) states, i.e., pairs of abstract states. (Compare this to Hoare logic, which is based on one instance of this idea: "types" of programs are implications between predicates on states). We show that this quite basic idea leads to very simple type systems with smooth meta-theory which, we hope, will find applications in the area of certified software.

Our approach is in strong contrast to works where authors have had the idea that types for imperative programs can be single abstract states, eg. the classic work by Volpano et al. on secure information flow [15]. They may have drawn their intuitions from the fact that, in many languages, the data-types of program variables are required to stay invariant during program runs. But data-types and languages where variables cannot migrate from one data-type to another are more of an exception than a rule in the big picture of imperative languages and notions of abstract state. Most kinds of abstract states or types (such as which variables are live etc.) do change during runs. Because of the better notion of type, our type systems allow us to infer and state sharper properties of programs than type systems based on the idea that a type is a single abstract state.

More technically, our achievement and contribution in this paper is the following. We introduce a framework for defining data-flow analyses that caters, besides the most classical data-flow analyses such as reaching definitions or very busy expressions, also for abstract-interpretation inspired analyses such

as constant propagation and analyses based on non-standard semantics of conditional constructs such as secure information flow. We then introduce a method for deriving a type system from an analysis defined in the framework and show that the type system obtained is always equivalent to the analysis. What the method does is very simple: in fact we only introduce two schematic type systems, one for all forward analyses and one for all backward analyses definable in the framework. A type system for a concrete analysis is obtained by mechanical instantiation of the appropriate schematic type system with the definition of the analysis. We look at a number of example analyses, most prominently secure information flow. We also discuss extraction of certificates (essential information) from typing derivations and lightweight, certificate-assisted type-checking that should be useful in the realm of certified code.

The paper is structured as follows. In Sec. 2, we introduce our framework for defining data-flow analyses. In Sec. 3, we present our method of deriving a type system from the definition of an analysis. In Sec. 4, we substantiate our development with examples. In Sec. 5, we discuss certificate generation from type derivations and certificate-assisted type-checking. Sec. 6 overviews the related work and Sec. 7 presents our conclusions.

2 A Framework for Data-Flow Analyses

We consider a simple imperative programming language (the WHILE-language). Its syntax is given by the following grammar:

$$P ::= p \mid P_1; P_2 \mid \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2 \mid \mathit{while}\ b\ \mathit{do}\ P_1$$

Here p is a primitive statement and b a Boolean expression (whose evaluation does not change the state of the program), similar to [7]. Primitive statements may be assignments or *skip*-statements; their details do not concern us here. Let **Stm** be the set of all programs.

Data-flow analyses work by calculating an assignment of abstract states to the edges of a control-flow graph. Depending on whether the analysis is forward or backward, either the direct or inverse control-flow graph of a given program is used and the calculation happens by propagating abstract state across the nodes of the graph in the appropriate direction.

We consider two types of data-flow analyses: forward may-analyses (based on least fixedpoint computation on the direct control-flow graph of a program) and backward must-analyses (based on greatest fixedpoint computation on the inverse control-flow diagram). The reasons to choose namely these two kinds as primitive will be explained in the next section. Forward must-analyses and backward may-analyses are variants of obtained by reversing the partial order on the state space.

For forward may-analyses, we require that our abstract state space $(D, \sqsubseteq)$ is an ω-complete upper semilattice. We assume that each primitive statement p is given a denotation $[\![p]\!] : D \to D$ (sending an abstract state before running p to the abstract state after). The function $[\![p]\!]$ must be monotone and upward ω-continuous. Also, two denotations $[\![b]\!]_{\mathsf{t}}, [\![b]\!]_{\mathsf{f}} : D \to D$ are associated with each expression b. If d is an abstract state before evaluating b, then $[\![b]\!]_{\mathsf{t}}(d)$ [resp. $[\![b]\!]_{\mathsf{f}}(d)$] correspond to the abstract states after evaluating to true, resp. false. After evaluating b the result of the evaluation is known and these denotations can add that knowledge to their arguments. The functions $[\![b]\!]_{\mathsf{t}}$ and $[\![b]\!]_{\mathsf{f}}$ must be monotone and upward ω-continuous as well. Often they are also reductive (i.e., $[\![b]\!]_{\mathsf{t}}(d) \sqsubseteq d \sqsupseteq [\![b]\!]_{\mathsf{f}}(d)$).

There is also a semantic function $\mathsf{combine} : D \times D \to D$ for combining the abstract state from before running a conditional statement (a *if*-statement or *while*-statement) into the abstract state immediately after that statement; it must be monotone and upward ω-continuous as well. The abstract states after the branches are combined using the join operation of D as usual, but for some analyses this simple joining is not enough. If the goal of an analysis is to determine the properties of program points (all kinds of value analyses are in this class), then the simple joining suffices because, at the confluence point, we have to take the disjunction of the properties on branches and this corresponds to the join on D. The situation is the same for the analyses that attempt to determine the properties of program paths (for example, where certain variables are used or defined)—disjunction (or join) has to be taken at confluence points. However, there are some analyses, most notably analyses for information flow, that do not belong to either class. These analyses do not attempt to find properties, but to make sure that all program runs are somehow similar (and to find the level of similarity). Here the similarity of all runs over one program path, and the similarity of all runs over the other program path does not imply the similarity of all runs after these paths have merged. Some extra processing is necessary and we have found that a form of post-processing after joining the information from two branches suffices. This post-processing function $\mathsf{combine}$ needs some extra information, namely the analysis information from the point before the branching.

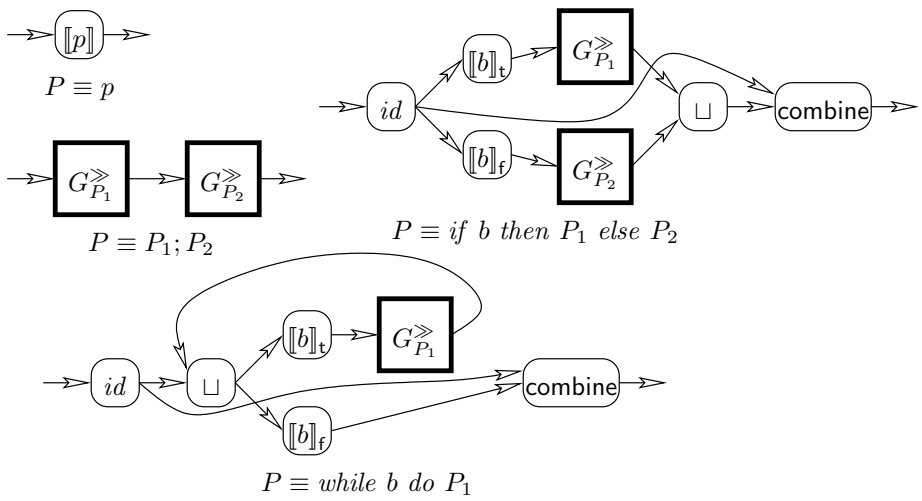

Figure 1: Schematic control-flow graph $G^{\gg}_P$ of a program P for forward may-analyses

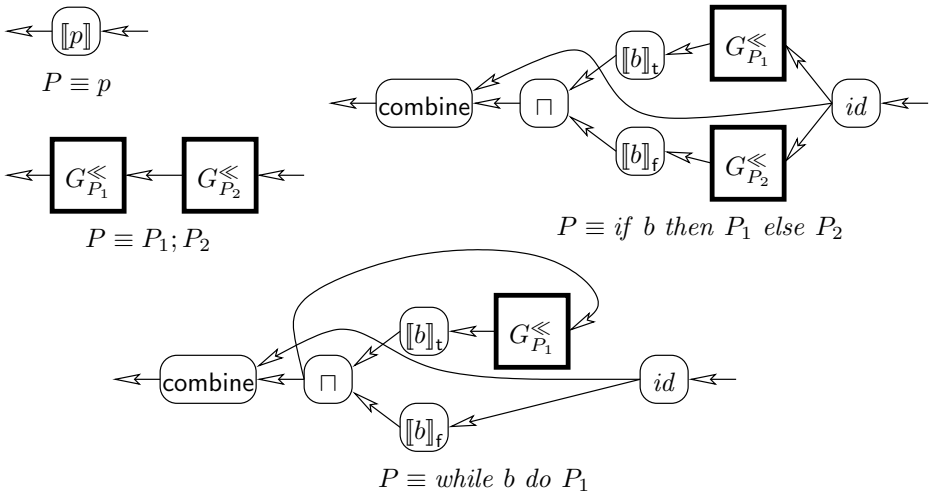

Figure 2: Schematic control-flow graph $G^{\ll}_P$ of a program P for backward must-analyses

For backward must-analyses, the abstract state space $(D, \sqsubseteq)$ must be an ω-complete lower semilattice. Again we must have functions $[\![p]\!], [\![b]\!]_\mathsf{t}, [\![b]\!]_\mathsf{f} : D \to D$, this time sending an abstract state after running p or evaluating b to true or false to the abstract state before, and a function $\mathsf{combine} : D \times D \to D$, which is used to combine the abstract state after running a conditional to the abstract state before. All these must be monotone and downward ω-continuous.

A control-flow graph G is always a tuple $(V, \mathrm{ar}, \ell, E, e^\mathrm{i}, e^\mathrm{f}, \sigma, \tau)$ where V is a set of *nodes*, $\mathrm{ar} : V \to \mathbb{N}$ gives the *indegree* of a node (the number of incoming edges), $\ell : (v : V) \to (D^{\mathrm{ar}(v)} \to D)$ maps a node to a corresponding *transfer function*, E is a set of *edges*, $e^\mathrm{i}, e^\mathrm{f} \in E$ are two distinguished edges called the *initial and final edge*, respectively, $\sigma : E \backslash \{e^\mathrm{i}\} \to V$ gives the *source* node of an edge and $\tau : E \backslash \{e^\mathrm{f}\} \to V \times \mathbb{N}$ gives the *target* node of an edge together with its *sequence number* among the incoming edges of that node. Note that the initial edge does not have a source and the final edge does not have a target. The function τ is injective—merging of abstract values may only be done by the transfer functions. Hence there exists an inverse partial function $\tau^{-1} : V \times \mathbb{N} \hookrightarrow E \backslash \{e^\mathrm{f}\}$.

The direct and inverse control-flow graphs $G^{\gg}_P$ and $G^{\ll}_P$ of a program P are defined inductively over the structure of P, the definitions are given in Fig. 1 and Fig. 2. The sets of nodes and edges, as well as the mappings ar, σ and τ and the edges e^i and e^f are presented there in a straightforward manner. The transfer functions are depicted inside the nodes; id denotes the identity function over D. The incoming edges of a node are numbered from top to bottom.

Abstract denotations are associated to arbitrary programs by a forward or a backward analysis in the

$$\frac{d_1 \sqsubseteq d_1' \quad \vdash_{\gg} P : d_1' \longrightarrow d_2' \quad d_2' \sqsubseteq d_2}{\vdash_{\gg} P : d_1 \longrightarrow d_2} \text{ (sub)}$$

$$\frac{}{\vdash_{\gg} p : d \longrightarrow [\![p]\!](d)} \text{ (prim)}$$

$$\frac{\vdash_{\gg} P_1 : d \longrightarrow d' \quad \vdash_{\gg} P_2 : d' \longrightarrow d''}{\vdash_{\gg} P_1; P_2 : d \longrightarrow d''} \text{ (seq)}$$

$$\frac{\vdash_{\gg} P_1 : [\![b]\!]_{\mathsf{t}}(d) \longrightarrow d' \quad \vdash_{\gg} P_2 : [\![b]\!]_{\mathsf{f}}(d) \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow \mathsf{combine}(d, d')} \text{ (if)}$$

$$\frac{\vdash_{\gg} P : [\![b]\!]_{\mathsf{t}}(d') \longrightarrow d' \quad d \sqsubseteq d'}{\vdash_{\gg} \mathit{while}\ b\ \mathit{do}\ P : d \longrightarrow \mathsf{combine}(d, [\![b]\!]_{\mathsf{f}}(d'))} \text{ (while)}$$

Figure 3: Schematic type system corresponding to forward may-analyses

following way. For any control graph G, we define a functional $\Phi_G : D \to ((E \to D) \to (E \to D))$ by

$$\Phi_G(d)(\iota)(e) = \begin{cases} d & \text{if } e = e^{\mathrm{i}} \\ \ell(v)(\iota(e_1), \ldots, \iota(e_{\mathrm{ar}(v)})), & \text{otherwise} \end{cases} \tag{1}$$

where $v = \sigma(e)$ and $e_i = \tau^{-1}(v, i)$.

The abstract denotation $[\![P]\!]_{\gg} : D \to D$ of a program P defined by a forward analysis is $[\![P]\!]_{\gg}(d) = \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e^{\mathrm{f}}_{G_P^{\gg}})$, where the least fixedpoint exists since $E \to D$ is an ω-complete upper semilattice and $\Phi_{G_P^{\gg}}(d)$ is upward ω-continuous. $[\![P]\!]_{\gg}$ is easily verified to be monotone and upward ω-continuous.

The abstract denotation $[\![P]\!]_{\ll} : D \to D$ assigned by a backward analysis to a program P is $[\![P]\!]_{\ll}(d) = \mathit{gfp}(\Phi_{G_P^{\ll}}(d))(e^{\mathrm{f}}_{G_P^{\ll}})$; the greatest fixedpoint exists since $E \to D$ is an ω-complete lower semilattice and $\Phi_{G_P^{\gg}}(d)$ is downward ω-continuous. $[\![P]\!]_{\ll} : D \to D$ is also monotone and downward ω-continuous.

3 Type systems

We are now ready to proceed to our type systems. We present a schematic type system for both forward and backward analyses. Type systems for concrete analyses are obtained as instances.

In both cases, types are of the form $d_1 \longrightarrow d_2$ where $d_1, d_2 \in D$, reflecting the idea that denotations of programs are transformers of (standard or non-standard) states. The intended meaning of a typing judgement $P : d_1 \longrightarrow d_2$ is that if the state before running P can be described by d_1, then the state after is described by d_2. This is independent of the direction of the analysis, i.e., the implication sign in the type always indicates the direction of running the program.

Also in both cases, the type system is extracted from the definition of control-flow graphs for programs. The typing rule for each statement-construct is read off from the corresponding clause in the definition of control-flow graphs. The rule (sub) is a subsumption rule combining the usual subsumption rule with structural subtyping for implications.

The type system derived from a forward analysis is presented in Fig. 3. All rules, except (while), are self-explanatory. In (while), d' is a form of a loop invariant—it corresponds to the abstract state at the back-edge in the direct control-flow graph of the while-statement. The normal use of the type system for analysing a program is this: given a program P and an abstract state d, find an abstract state d' such that $P : d \longrightarrow d'$. Inspecting the rule set, one can see that the process is immediate except for two aspects: (1) it is not *a priori* clear if and when it is necessary to apply (sub) and (2) applying (while) takes guessing a suitable invariant.

The type system is equivalent to the analysis in the sense of the following theorem.

Theorem 1. *For any $P \in$ **Stm** and $d_1, d_2 \in D$ holds: $\vdash_{\gg} P : d_1 \longrightarrow d_2$ iff $[\![P]\!]_{\gg}(d_1) \sqsubseteq d_2$.*

We see that, given an initial abstract state, the type system can approve as final any abstract state that is weaker (containing more concrete states) than the final abstract state computed by the analysis. The analysis gives the strongest (smallest) postcondition of a precondition while the type system accepts any postcondition.

The type system corresponding to backward analyses is given in Fig. 4. (Remember that the direction of the arrow still indicates the direction of running the program.) The rules are different, since inverse

$$\frac{d_1 \sqsubseteq d_1' \quad \vdash_{\ll} P : d_1' \longrightarrow d_2' \quad d_2' \sqsubseteq d_2}{\vdash_{\ll} P : d_1 \longrightarrow d_2} \text{ (sub)}$$

$$\frac{}{\vdash_{\ll} p : [\![p]\!](d) \longrightarrow d} \text{ (prim)}$$

$$\frac{\vdash_{\ll} P_1 : d'' \longrightarrow d' \quad \vdash_{\ll} P_2 : d' \longrightarrow d}{\vdash_{\ll} P_1; P_2 : d'' \longrightarrow d} \text{ (seq)}$$

$$\frac{\vdash_{\ll} P_1 : d_{\mathrm{t}} \longrightarrow d \quad \vdash_{\ll} P_2 : d_{\mathrm{f}} \longrightarrow d}{\vdash_{\ll} \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : \mathsf{combine}(d, [\![b]\!]_{\mathrm{t}}(d_{\mathrm{t}}) \sqcap [\![b]\!]_{\mathrm{f}}(d_{\mathrm{f}})) \longrightarrow d} \text{ (if)}$$

$$\frac{\vdash_{\ll} P : d' \longrightarrow [\![b]\!]_{\mathrm{t}}(d') \sqcap [\![b]\!]_{\mathrm{f}}(d)}{\vdash_{\ll} \mathit{while}\ b\ \mathit{do}\ P : \mathsf{combine}(d, [\![b]\!]_{\mathrm{t}}(d') \sqcap [\![b]\!]_{\mathrm{f}}(d)) \longrightarrow d} \text{ (while)}$$

Figure 4: Schematic type system corresponding to backward must-analyses

control-flow graphs differ from direct ones. But the ideas are still exactly the same. Here the loop invariant d' corresponds to the abstract state at the back-edge of the inverse control-flow graph of the while-statement. The normal use of the type system to analyze a program is this: given a program P and an abstract state d, find an abstract state d' such that $P : d' \longrightarrow d$.

Again we have a perfect match between the type system and the analysis:

Theorem 2. *For any $P \in \mathbf{Stm}$ and $d_1, d_2 \in D$ holds: $\vdash_{\ll} P : d_1 \longrightarrow d_2$ iff $d_1 \sqsubseteq [\![P]\!]_{\ll}(d_2)$.*

Here the type system can identify as initial any abstract state that is stronger than the initial state computed by the analysis. While the analysis delivers the weakest precondition of a postcondition, the type system approves any precondition.

It is now also clear why we wanted forward analyses to be "may" and backward analyses to be "must": this gives the subsumption rules the normal shape: the implication construct is covariant in its succedent and contravariant in the antecedent and not the other way around.

4 Examples

To demonstrate the generality of the framework given in Sec. 2 and the mechanics of our type systems, we now show how some concrete data-flow analyses can be expressed within it and discuss the type systems that our method delivers for them. We present the analyses for constant propagation, very busy expressions, and secure information flow. The appendix contains subsequent examples.

In the examples below, we assume that the programming language is defined over a supply **Var** of arithmetic variables, that primitive statements p have the form $x := \mathit{aop}(x_1, \ldots, x_k)$ where $x, x_1, \ldots, x_k \in \mathbf{Var}$ and *aop* is an arithmetic operator with arity k, and Boolean expressions have the form $\mathit{bop}(x_1, \ldots, x_k)$ where *bop* is an Boolean operator with arity k. In the following we let o range over both *aop* and *bop*.

4.1 Constant Propagation

The aim of constant propagation is to determine for all program points the values of as many variables as possible. In the context of type systems, we are given the initial values of some variables and want to determine the final values of as many variables as possible.

Let $\mathbb{Z}^\top = \mathbb{Z} \dot\cup \{\top\}$, where $\top$ is the largest element and other elements are incomparable. The underlying lattice D for the constant propagation is $(\mathbf{Var} \to \mathbb{Z}^\top)_\bot$ where $X_\bot$ denotes the set $X \dot\cup \{\bot\}$ with $\bot$ the smallest element and all other elements ordered in the same way as in X, and the order on $\mathbf{Var} \to \mathbb{Z}^\top$ is defined pointwise.

The analysis works in the forward direction. For the ease of the exposition let $\mathit{aop} \in \mathbb{Z} \cup \{+, \times\}$ and $\mathit{bop} \in \{\leq, =\}$. Here the elements of $\mathbb{Z}$ are nullary operators and the rest are binary. If $d = \bot$ then $[\![x := o(x_1, \ldots, x_k)]\!](d) = \bot$ and $[\![o(x_1, \ldots, x_k)]\!]_{\mathrm{t}}(d) = [\![o(x_1, \ldots, x_k)]\!]_{\mathrm{f}}(d) = \bot$. Otherwise

$$[\![x := n]\!](d) = d[x \mapsto n]$$

$$[\![x := y + z]\!](d) = \begin{cases} d[x \mapsto d(y) + d(z)], & d(y), d(z) \in \mathbb{Z} \\ d[x \mapsto \top], & \text{otherwise} \end{cases}$$

$$[\![x := y \times z]\!](d) = \begin{cases} d[x \mapsto 0], & d(y) = 0 \text{ or } d(z) = 0 \\ d[x \mapsto d(y) \cdot d(z)], & d(y), d(z) \in \mathbb{Z}\backslash\{0\} \\ d[x \mapsto \top], & \text{otherwise} \end{cases}$$

$$[\![x \leq y]\!]_{\mathsf{t}}(d) = \begin{cases} \bot, & d(x), d(y) \in \mathbb{Z} \text{ and } d(x) > d(y) \\ d, & \text{otherwise} \end{cases}$$

$$[\![x \leq y]\!]_{\mathsf{f}}(d) = \begin{cases} \bot, & d(x), d(y) \in \mathbb{Z} \text{ and } d(x) \leq d(y) \\ d, & \text{otherwise} \end{cases}$$

$$[\![x = y]\!]_{\mathsf{t}}(d) = \begin{cases} \bot, & d(x), d(y) \in \mathbb{Z} \text{ and } d(x) \neq d(y) \\ d[x \mapsto d(y)], & d(y) \in \mathbb{Z} \text{ and } d(x) = \top \\ d[y \mapsto d(x)], & d(x) \in \mathbb{Z} \text{ and } d(y) = \top \\ d, & \text{otherwise} \end{cases}$$

$$[\![x = y]\!]_{\mathsf{f}}(d) = \begin{cases} \bot, & d(x), d(y) \in \mathbb{Z} \text{ and } d(x) = d(y) \\ d, & \text{otherwise} \end{cases}$$

Finally, $\mathsf{combine}(d, d') = d'$. The corresponding type system is given in Fig. 5. We have presented the different cases in the definition of semantics of primitive statements and Boolean expressions as different rules. We see that sometimes we do not have to type some subprograms. The reason is that in this case according to Fig. 3 this subprogram has to have a type of the form $\bot \longrightarrow d$, but $\not\vdash_{\gg} P : \bot \longrightarrow \bot$ can be easily inferred for any $P \in \mathbf{Stm}$.

4.2 Very Busy Expressions

The aim of this analysis is to determine for each program point, which expressions will certainly be computed after the execution has passed that point and before any of the variables in these expressions are redefined [10]. The results of this analysis may be used to decide whether it is worthwhile to precompute some expressions.

Let **Exp** be the set of all expressions, i.e. terms $o(x_1, \ldots, x_k)$. The underlying lattice D for very busy expressions analysis is $\mathcal{P}(\mathbf{Exp})$ (the set of all subsets of **Exp**) with the natural order. For each $x \in \mathbf{Var}$ let $\mathbf{Exp}_x \subseteq \mathbf{Exp}$ be the set of all these expressions that contain x. The analysis works in the backward direction. We define

$$[\![x := o(x_1, \ldots, x_k)]\!](d) = (d \backslash \mathbf{Exp}_x) \cup \{o(x_1, \ldots, x_k)\}$$

$$[\![o(x_1, \ldots, x_k)]\!]_{\mathsf{t}}(d) = [\![o(x_1, \ldots, x_k)]\!]_{\mathsf{f}}(d) = d \cup \{o(x_1, \ldots, x_k)\}$$

and $\mathsf{combine}(d, d') = d'$. This gives us the analysis in [10]. The equivalent type system is given in Fig. 6.

4.3 Secure Information Flow

Let $\mathcal{L} = (\mathcal{L}, \vee.\wedge, \mathbf{1}, \mathbf{0})$ be a finite lattice of *security classes*. At the initial and final states of the program, a security class is associated with each variable. The goal of the analysis is to verify that the initial value of some variable x may only affect the final value of the variable y if the final security class of y is at least as great as the initial security class of x.

Let $\mathbf{Var}' = \mathbf{Var} \dot\cup \{\mathsf{pc}\}$. The underlying lattice D of the analysis is $\mathbf{Var}' \to \mathcal{L}$ where the ordering is defined pointwise. Given $d \in D$ describing the security classes at the start of the program the program analysis attempts to find for each variable the security class that upper bounds the information that may flow to its final value. The security class $d(\mathsf{pc})$ contains an upper bound of the information that was used to make the decisions (at branching points) that caused the execution to reach the current program point; $d(\mathsf{pc})$ should be equal to $\mathbf{0}$ at the start of the program. The notation pc is a mnemonic for "program counter".

The analysis works in the forward direction. The semantics of statements and expressions is defined by

$$[\![x := o(x_1, \ldots, x_k)]\!](d) = d[x \mapsto d(x_1) \vee \cdots \vee d(x_k) \vee d(\mathsf{pc})]$$

$$[\![o(x_1, \ldots, x_k)]\!]_{\mathsf{t}}(d) = [\![o(x_1, \ldots, x_k)]\!]_{\mathsf{f}}(d) = d[\mathsf{pc} \mapsto d(x_1) \vee \cdots \vee d(x_k) \vee d(\mathsf{pc})] \ .$$

$$\frac{d_1 \sqsubseteq d_1' \quad \vdash_{\gg} P : d_1' \longrightarrow d_2' \quad d_2' \sqsubseteq d_2}{\vdash_{\gg} P : d_1 \longrightarrow d_2} \text{ (sub)}$$

$$\frac{}{\vdash_{\gg} x := o(x_1, \ldots, x_k) : d \longrightarrow d[x \mapsto \top]} \text{ (prim_1)}$$

$$\frac{}{\vdash_{\gg} x := n : d \longrightarrow d[x \mapsto n]} \text{ (prim_2)}$$

$$\frac{d(y) \in \mathbb{Z} \quad d(z) \in \mathbb{Z}}{\vdash_{\gg} x := o(y, z) : d \longrightarrow d[x \mapsto [\![o]\!](d(y), d(z))]} \text{ (prim_3)}$$

$$\frac{d(y) = 0 \text{ or } d(z) = 0}{\vdash_{\gg} x := y \times z : d \longrightarrow d[x \mapsto 0]} \text{ (prim_4)}$$

$$\frac{\vdash_{\gg} P_1 : d \longrightarrow d' \quad \vdash_{\gg} P_2 : d' \longrightarrow d''}{\vdash_{\gg} P_1; P_2 : d \longrightarrow d''} \text{ (seq)}$$

$$\frac{\vdash_{\gg} P_1 : d \longrightarrow d' \quad \vdash_{\gg} P_2 : d \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ o(x, y)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'} \text{ (if_1)}$$

$$\frac{d(x) \in \mathbb{Z} \quad d(y) \in \mathbb{Z} \quad [\![o]\!](d(x), d(y)) \quad \vdash_{\gg} P_1 : d \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ o(x, y)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'} \text{ (if_2)}$$

$$\frac{d(x) \in \mathbb{Z} \quad d(y) \in \mathbb{Z} \quad \neg[\![o]\!](d(x), d(y)) \quad \vdash_{\gg} P_2 : d \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ o(x, y)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'} \text{ (if_3)}$$

$$\frac{d(x) \in \mathbb{Z} \quad d(y) = \top \quad \vdash_{\gg} P_1 : d[y \mapsto d(x)] \longrightarrow d' \quad \vdash_{\gg} P_2 : d \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ x = y\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'} \text{ (if_4)}$$

$$\frac{d(y) \in \mathbb{Z} \quad d(x) = \top \quad \vdash_{\gg} P_1 : d[x \mapsto d(y)] \longrightarrow d' \quad \vdash_{\gg} P_2 : d \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ x = y\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'} \text{ (if_5)}$$

$$\frac{\vdash_{\gg} P : d' \longrightarrow d' \quad d \sqsubseteq d'}{\vdash_{\gg} \mathit{while}\ o(x, y)\ \mathit{do}\ P : d \longrightarrow d'} \text{ (while_1)}$$

$$\frac{\vdash_{\gg} P : d' \longrightarrow d' \quad d \sqsubseteq d' \quad d'(x) \in \mathbb{Z} \quad d'(y) \in \mathbb{Z} \quad [\![o]\!](d'(x), d'(y))}{\vdash_{\gg} \mathit{while}\ o(x, y)\ \mathit{do}\ P : d \longrightarrow \bot} \text{ (while_2)}$$

$$\frac{d(x) \in \mathbb{Z} \quad d(y) \in \mathbb{Z} \quad \neg[\![o]\!](d(x), d(y))}{\vdash_{\gg} \mathit{while}\ o(x, y)\ \mathit{do}\ P : d \longrightarrow d} \text{ (while_3)}$$

$$\frac{\vdash_{\gg} P : d'[y \mapsto d'(x)] \longrightarrow d' \quad d \sqsubseteq d' \quad d'(x) \in \mathbb{Z} \quad d'(y) = \top}{\vdash_{\gg} \mathit{while}\ x = y\ \mathit{do}\ P : d \longrightarrow d'} \text{ (while_4)}$$

$$\frac{\vdash_{\gg} P : d'[x \mapsto d'(y)] \longrightarrow d' \quad d \sqsubseteq d' \quad d'(y) \in \mathbb{Z} \quad d'(x) = \top}{\vdash_{\gg} \mathit{while}\ x = y\ \mathit{do}\ P : d \longrightarrow d'} \text{ (while_5)}$$

Figure 5: Type system from constant propagation analysis

$$\frac{d_1 \subseteq d_1' \quad \vdash_{\ll} P : d_1' \longrightarrow d_2' \quad d_2' \subseteq d_2}{\vdash_{\ll} P : d_1 \longrightarrow d_2} \text{ (sub)}$$

$$\frac{}{\vdash_{\ll} x := o(x_1, \ldots, x_k) : (d \backslash \mathbf{Exp}_x) \cup \{o(x_1, \ldots, x_k)\} \longrightarrow d} \text{ (prim)}$$

$$\frac{\vdash_{\ll} P_1 : d \longrightarrow d' \quad \vdash_{\ll} P_2 : d' \longrightarrow d''}{\vdash_{\ll} P_1; P_2 : d \longrightarrow d''} \text{ (seq)}$$

$$\frac{\vdash_{\ll} P_1 : d_{\mathrm{t}} \longrightarrow d \quad \vdash_{\ll} P_2 : d_{\mathrm{f}} \longrightarrow d}{\vdash_{\ll} \mathit{if}\ o(x_1, \ldots, x_k)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : (d_{\mathrm{t}} \cap d_{\mathrm{f}}) \cup \{o(x_1, \ldots, x_k)\} \longrightarrow d} \text{ (if)}$$

$$\frac{\vdash_{\ll} P : d' \longrightarrow (d' \cap d) \cup \{o(x_1, \ldots, x_k)\}}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : (d' \cap d) \cup \{o(x_1, \ldots, x_k)\} \longrightarrow d} \text{ (while)}$$

Figure 6: Type system from very busy expressions analysis

$$\frac{d_1 \sqsubseteq d_1' \quad \vdash_{\gg} P : d_1' \longrightarrow d_2' \quad d_2' \sqsubseteq d_2}{\vdash_{\gg} P : d_1 \longrightarrow d_2} \text{ (sub)}$$

$$\frac{}{\vdash_{\gg} x := o(x_1, \ldots, x_k) : d \longrightarrow d[x \mapsto d(x_1) \vee \cdots \vee d(x_k) \vee d(\mathsf{pc})]} \text{ (prim)}$$

$$\frac{\vdash_{\gg} P_1 : d \longrightarrow d' \quad \vdash_{\gg} P_2 : d' \longrightarrow d''}{\vdash_{\gg} P_1; P_2 : d \longrightarrow d''} \text{ (seq)}$$

$$\frac{\begin{array}{c}\vdash_{\gg} P_1 : d[\mathsf{pc} \mapsto d(x_1) \vee \cdots \vee d(x_k) \vee d(\mathsf{pc})] \longrightarrow d' \\ \vdash_{\gg} P_2 : d[\mathsf{pc} \mapsto d(x_1) \vee \cdots \vee d(x_k) \vee d(\mathsf{pc})] \longrightarrow d'\end{array}}{\vdash_{\gg} \mathit{if}\ o(x_1, \ldots, x_k)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'[\mathsf{pc} \mapsto d(\mathsf{pc})]} \text{ (if)}$$

$$\frac{\vdash_{\gg} P : d' \longrightarrow d' \quad d \sqsubseteq d' \quad \forall i : d'(x_i) \leq d'(\mathsf{pc})}{\vdash_{\gg} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d \longrightarrow d'[\mathsf{pc} \mapsto d(\mathsf{pc})]} \text{ (while)}$$

Figure 7: Type system from a forward information flow analysis

I.e. the security level of the affected quantity (a variable or the program counter) is updated with the join of security levels of affecting quantities.

We see that the explicit flows in the program (assigning a variable to another) are handled directly by the analysis. The implicit flows (one variable controlling whether something is assigned to another one) are handled by increasing of the security class of the program counter. At the points where the conditional statements end, the $\mathsf{combine}$-function restores the security class of pc to the one before the conditional statement: $\mathsf{combine}(d, d') = w'[\mathsf{pc} \mapsto d(\mathsf{pc})]$.

The resulting type system is given in Fig. 7. Actually, if we followed Fig. 3 to the letter then we would have had the following rule for *while*:

$$\frac{\vdash_{\gg} P : d'[\mathsf{pc} \mapsto d'(x_1) \vee \cdots \vee d'(x_k) \vee d'(\mathsf{pc})] \longrightarrow d' \quad d \sqsubseteq d'}{\vdash_{\gg} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d \longrightarrow d'[\mathsf{pc} \mapsto d(\mathsf{pc})]} \tag{2}$$

(in the consequent both $[\![o(x_1, \ldots, x_k)]\!]_{\mathsf{f}}$ and $\mathsf{combine}$ change the value of $d'(\mathsf{pc})$ but $\mathsf{combine}$ is the last one). To get the rule (while) in Fig. 7 we use the fact that for all programs P and all $d \in D$ we have $[\![P]\!]_{\gg}(d)(\mathsf{pc}) = d(\mathsf{pc})$. We could then replace d' to the right of the arrow in (2) by $d'[\mathsf{pc} \mapsto d'(x_1) \vee \cdots \vee d'(x_k) \vee d'(\mathsf{pc})]$ which we could then replace with d' in both sides of the arrow. But now d' can not be arbitrary, but $d'(\mathsf{pc})$ must be at least $d'(x_i)$ for all $i \in \{1, \ldots, k\}$.

5 Certification and Checking of Typings

In this section we describe how to convince another party that a program P has a given type $d_1 \longrightarrow d_2$. We assume that we know a type derivation T with the root $\vdash_{\gg} P : d_1 \longrightarrow d_2$ (we concentrate on the type system in Fig. 3, the system in Fig. 4 is handled similarly). The verifier only has to compare types and compute primitive semantic functions.

The *certificate (proof)* for $\vdash_{\gg} P : d_1 \longrightarrow d_2$ has the form $(d_1, \mathfrak{P}(P, T, d_1), d_2)$ with an easily constructible $\mathfrak{P}(P, T, d_1) \in D^*$. The function $\mathfrak{P}$ is defined below. Let R be the last rule in the type inference tree T for $\vdash_{\gg} P : d_1 \longrightarrow d_2$. Let T' be equal to T without the last rule (if the R is (seq) or (if) then there are two such trees, we denote them T_1' and T_2'). Let b, P_1, P_2, d' and d_1' have the same meaning as in the rules. If R is (while) then let P_1 denote the loop body. We define

$$\mathfrak{P}(P, T, d_1) = \begin{cases} \mathfrak{P}(P, T', d_1') & R \text{ is (sub)} \\ [] & R \text{ is (prim)} \\ \mathfrak{P}(P_1, T_1', d_1) \cdot \mathfrak{P}(P_2, T_2', d') & R \text{ is (seq)} \\ \mathfrak{P}(P_1, T_1', [\![b]\!]_{\mathsf{t}}(d_1)) \cdot \mathfrak{P}(P_2, T_2', [\![b]\!]_{\mathsf{f}}(d_1)) & R \text{ is (if)} \\ d' \cdot \mathfrak{P}(P_1, T', [\![b]\!]_{\mathsf{t}}(d')) & R \text{ is (while).} \end{cases}$$

Let us now describe the *checking (verification) algorithm* $\mathfrak{V}$. It takes as arguments a program P, an abstract state d_1' and a certificate $\mathsf{p} \in D^*$ and either fails or returns an abstract state d_2' and $\mathsf{p}' \in D^*$. Failure implies that the proof is not accepted. To verify the proof (d_1, p, d_2) for the program P, one invokes $\mathfrak{V}(P, d_1, \mathsf{p})$, receives (d_2', p') and accepts if $d_2' \leq d_2$. The function $\mathfrak{V}$ is defined in Fig. 8. Here

case P **of**

p:	**return** $([\![p]\!](d), \mathsf{p})$
$P_1; P_2$:	$(d', \mathsf{p}') \leftarrow \mathfrak{V}(P_1, d, \mathsf{p})$ **return** $\mathfrak{V}(P_2, d', \mathsf{p}')$
if b *then* P_1 *else* P_2:	$(d_1, \mathsf{p}') \leftarrow \mathfrak{V}(P_1, [\![b]\!]_\mathsf{t}(d), \mathsf{p})$ $(d_2, \mathsf{p}'') \leftarrow \mathfrak{V}(P_2, [\![b]\!]_\mathsf{f}(d), \mathsf{p}')$ **return** $(\mathsf{combine}(d, d_1 \sqcup d_2), \mathsf{p}'')$
while b *do* P_1:	$d' \leftarrow \mathbf{head}(\mathsf{p})$ $\mathbf{check}(d' \sqsupseteq d)$ $(d'', \mathsf{p}') \leftarrow \mathfrak{V}(P_1, [\![b]\!]_\mathsf{t}(d'), \mathbf{tail}(\mathsf{p}))$ $\mathbf{check}(d'' \sqsubseteq d')$ **return** $(\mathsf{combine}(d, [\![b]\!]_\mathsf{f}(d')), \mathsf{p}')$

Figure 8: The verification algorithm $\mathfrak{V}(P, d, \mathsf{p})$

check checks that its argument is true, and fails otherwise. The function **head** also fails on empty input.

Theorem 3. *If* $\vdash\!\!\!\gg P : d_1 \longrightarrow d_2$ *is derivable and* T *is the corresponding derivation, then the invocation of* $\mathfrak{V}(P, d_1, \mathfrak{P}(P, T, d_1))$ *does not fail, and the returned pair* (d_2', p') *satisfies* $d_2' \sqsubseteq d_2$.

Theorem 4. *If* p *is such that* $\mathfrak{V}(P, d_1, \mathsf{p}) = (d_2, \mathsf{p}')$, *then* $\vdash\!\!\!\gg P : d_1 \longrightarrow d_2$.

The checking algorithm $\mathfrak{V}$ actually "reads" the certificate p only if it encounters a *while*-loop in the program. It must remember the value d' as long as it verifies the loop body because the value of d' is used in a check after that. If space at the checking device is at premium then the proof may repeat the loop invariant d' after the proof for the loop body. I.e., we define $\mathfrak{P}^\star(\textit{while } b \textit{ do } P_1, T, d_1) = d' \cdot \mathfrak{P}^\star(P_1, T', [\![b]\!]_\mathsf{t}(d')) \cdot d'$ for the (while)-rule and as $\mathfrak{P}$ for other rules. Now $\mathfrak{V}$ no longer has to store d' while it verifies the loop body. But $\mathfrak{P}^\star$ may cheat—it may make the two copies of d' different. To avoid this we let $\mathfrak{V}$ store a *cryptographic message digest* of the first copy of d' and compare it to the second copy. Although such solution does not eliminate the storage requirements completely, it considerably reduces them—a digest will typically require 160 to 256 bits, considerably less than the entire d'.

Let $H : \{0,1\}^* \to \{0,1\}^k$ be a *collision-resistant hash function* where k is typically 160 or 256 (as the arguments of H are arbitrary bit-strings, we assume that H can be applied to anything representable). Intuitively (see [14] for precise definitions) this means that it is very hard to find *collisions* for H—pairs (x_1, x_2) where $x_1 \neq x_2$ but $H(x_1) = H(x_2)$. Let $\mathfrak{V}^H(\textit{while } b \textit{ do } P_1, d, \mathsf{p})$ be the following algorithm

$$d_1 \leftarrow \mathbf{head}(\mathsf{p}); \mathbf{check}(d_1 \sqsupseteq d); h \leftarrow H(d_1); (d_2, \mathsf{p}') \leftarrow \mathfrak{V}^H(P_1, [\![b]\!]_\mathsf{t}(d_1), \mathbf{tail}(\mathsf{p}));$$
$$d_3 \leftarrow \mathbf{head}(\mathsf{p}'); \mathbf{check}(h = H(d_3)); \mathbf{check}(d_2 \sqsubseteq d_3); \mathbf{return}\ (\mathsf{combine}(d, [\![b]\!]_\mathsf{f}(d_3)), \mathbf{tail}(\mathsf{p}'))$$

and let $\mathfrak{V}^H$ be defined like $\mathfrak{V}$ for other programs. Using more advanced integrity structures (authenticated search trees [3]), the space requirement per loop may be decreased even more (to the size of a single identifier, 2 or 4 bytes).

Theorem 5. *There exists a polynomial-time algorithm* K *such that, if* p, P *and* d_1 *are such that* $\mathfrak{V}^H(P, d_1, \mathsf{p}) = d_2$, *then either* $\vdash\!\!\!\gg P : d_1 \longrightarrow d_2$ *or* $K^H(P, d_1, \mathsf{p})$ *outputs a collision for* H.

6 Related work

The previous work on relating program analyses and type systems has been concentrating on control-flow analyses (analyses that attempt to show that the program does not make an illegal step) [11, 5, 12, 2]. These type systems have a more complex structure than the ones considered here and typically feature universally and existentially type-variable quantified implications (big meets and joins of implications). These features allow these type systems to have principal types; the type of a program fragment typically contains as much information as the denotational abstract semantics of that fragment. In contrast, a type in our type system corresponds to just one point (pair of abstract states) in the graph of the abstract semantics.

The closest to our work is probably the work of Naik and Palsberg [7]. They consider the same language as we do and their abstract semantics can be cast in our terms as well. But they only consider

the case where the underlying lattice D is a powerset (and they do not consider combine-functions, which means combine-function are just second projections). And similarly to the systems mentioned above, they quantify over type variables in implications and a type again corresponds to the entire semantics.

The certification of typing of Sec. 5 is reminiscent of Necula's and Lee's proof-carrying code [8]. Whereas the preliminary versions included the whole proof (derivation tree) in the certificate, later work [9] has concentrated on including only the information that is expensive to determine; this is similar to only including some typing information for loops. The same approach is present in the *lightweight bytecode verification* of Rose [13]—a control-flow graph based approach where the labels of all back-edges are included in the certificate. An implementation and soundness and completeness proofs of this approach are given in [6].

The analyses for secure information flow presented in this paper, as well as the resulting type systems are similar to those of Amtoft and Banerjee [1], and more precise than those by Volpano et al. [15]. The reason is that, for us, a type is a pair of security class assignments to variables (corresponding to their security classes before and after a program run), while they insist (for reasons that escape us) that security classes of variables are invariant.

7 Conclusions and Future Work

We have shown how imperative data-flow analyses definable in a certain framework can be equivalently cast as type systems, proceeding from a very basic idea about what a type of an imperative program ought to be. The resulting type systems are, to our judgement, extremely intuitive and well-motivated, so we believe we have once more witnessed how useful it can be to have the basic concepts right in any theory. We also believe our type systems are practically applicable in certified software.

As future work, we plan to further explore the limits of our analysis framework (especially the applications of combine-functions) and see if the framework can be liberalized in interesting ways while keeping the type systems simple. We also plan to study logics corresponding to more fine-grained types of analyses.

References

[1] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2004.

[2] Torben Amtoft and Franklyn A. Turbak. Faithful translations between polyvariant flows and polymorphic types. In Gert Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2000.

[3] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Eliminating counterevidence with applications to accountable certificate management. *Journal of Computer Security*, 10(3):273–296, 2002.

[4] Patrick Cousot. Types as abstract interpretations. In *POPL*, pages 316–331, 1997.

[5] Nevin Heintze. Control-flow analysis and type systems. In Alan Mycroft, editor, *SAS*, volume 983 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 1995.

[6] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001.

[7] Mayur Naik and Jens Palsberg. A type system equivalent to a model checker. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 374–388. Springer, 2005.

[8] George C. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.

[9] George C. Necula and Shree Prakash Rahul. Oracle-based checking of untrusted software. In *POPL*, pages 142–154, 2001.

[10] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[11] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst.*, 17(4):576–599, 1995.

[12] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 22(3):263–317, 2001.

[13] Eva Rose. Lightweight bytecode verification. *Journal of Automated Reasoning*, 31:303–334, 2003.

[14] Douglas R. Stinson. *Cryptography Theory and Practice, 2nd ed.* CRC Press, 2002.

[15] Dennis M. Volpano, Geoffrey Smith, and Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

$$\frac{d_1 \sqsubseteq d_1' \quad \vdash_{\ll} P : d_1' \longrightarrow d_2' \quad d_2' \sqsubseteq d_2}{\vdash_{\ll} P : d_1 \longrightarrow d_2} \text{ (sub)}$$

$$\frac{}{\vdash_{\ll} x := o(x_1, \ldots, x_k) : d[x_i \mapsto d(x_i) \wedge d(x)]_{i=1}^{k}[\mathsf{pc} \mapsto d(\mathsf{pc}) \wedge d(x), x \mapsto \mathbf{1}] \longrightarrow d} \text{ (prim)}$$

$$\frac{\vdash_{\ll} P_1 : d'' \longrightarrow d' \quad \vdash_{\ll} P_2 : d' \longrightarrow d}{\vdash_{\ll} P_1; P_2 : d'' \longrightarrow d} \text{ (seq)}$$

$$\frac{\vdash_{\ll} P_1 : d' \longrightarrow d \quad \vdash_{\ll} P_2 : d' \longrightarrow d}{\vdash_{\ll} \mathit{if}\ o(x_1, \ldots, x_k)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^{k}[\mathsf{pc} \mapsto d(\mathsf{pc})] \longrightarrow d} \text{ (if)}$$

$$\frac{\vdash_{\ll} P : d' \longrightarrow d' \quad d' \sqsubseteq d \quad \forall i : d'(x_i) \leq d'(\mathsf{pc})}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d'[\mathsf{pc} \mapsto d(\mathsf{pc})] \longrightarrow d} \text{ (while)}$$

Figure 9: Type system from a backward information flow analysis

A Subsequent Example Analyses

A.1 A Backwards Secure Information Flow Analysis

The domain D of a backwards secure information flow analysis is the same as that of the forwards analysis. Given $d \in D$ describing the security classes at the end of the program the program analysis attempts to find for each variable the maximum initial security class, such that the initial value of that variable does not flow to the final value of any variable with a lower security class. The security class $w(\mathsf{pc})$ holds a similar upper bound for the decisions influencing whether an assignment to a variable is executed or not; it should equal $\mathbf{1}$ at the end of the program (i.e. at the beginning of the analysis).

Let us define an auxiliary function $la : \mathcal{P}(\mathbf{Var}') \times D \times \mathcal{L} \to D$ by

$$la(X, d, c)(x) = \begin{cases} d(x) \wedge c, & x \in X \\ d(x), & x \notin X \end{cases} .$$

I.e. la is used to lower the security classes of the variables in X to the level of at most c. The semantics of statements $[\![x := o(x_1, \ldots, x_k)]\!]$ is defined by

$$[\![x := o(x_1, \ldots, x_k)]\!](d) = la(\{x_1, \ldots, x_k, \mathsf{pc}\}, d, d(x))[x \mapsto \mathbf{1}] \ .$$

I.e. before the statement $x := o(x_1, \ldots, x_k)$ the variable x is dead hence the secrecy of its value is of no importance. As the values of x_i may influence the final values of the variables, they cannot be more secret than x is after the statement. The same holds for implicit flows through pc. The semantics of Boolean expressions is

$$[\![o(x_1, \ldots, x_k)]\!]_{\mathsf{t}}(d) = [\![o(x_1, \ldots, x_k)]\!]_{\mathsf{f}}(d) = la(\{x_1, \ldots, x_k\}, d, d(\mathsf{pc})) \ .$$

and $\mathsf{combine}(d, d') = d'[\mathsf{pc} \mapsto d(\mathsf{pc})]$. So the variables x_i affecting the choice of the branch cannot be any more secret than the implicit flows inside the branches. Before the conditional statement, the implicit flows through pc are the same as after (and smaller than inside the branches), so the $\mathsf{combine}$-function updates $d(\mathsf{pc})$.

The type system derived from this analysis is given in Fig. 9. Well, if we had followed Fig. 4 to the letter then the rule for *while*-statements would have been

$$\frac{\vdash_{\ll} P : d' \longrightarrow (d' \sqcap d)[x_i \mapsto d'(x_i) \wedge d(x_i) \wedge d'(\mathsf{pc}) \wedge d(pc)]_{i=1}^{k}}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : (d' \sqcap d)[x_i \mapsto d'(x_i) \wedge d(x_i) \wedge d'(\mathsf{pc}) \wedge d(pc)]_{i=1}^{k}[\mathsf{pc} \mapsto d(\mathsf{pc})] \longrightarrow d}$$

Here we'll rename $d' \sqcap d$ to d'. The original d' is then just something that is larger than the new d'. As the original d' only occurs at the left side of the arrow in the premise we can safely replace it with the new d'. We also have to record that the new d' is smaller than d. This gives us

$$\frac{\vdash_{\ll} P : d' \longrightarrow d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^{k} \quad d' \sqsubseteq d}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^{k}[\mathsf{pc} \mapsto d(\mathsf{pc})] \longrightarrow d}$$

$$\frac{\phi_1 \Rightarrow \phi_1' \quad \vdash_{\ll} P : \phi_1' \longrightarrow \phi_2' \quad \phi_2' \Rightarrow \phi_2}{\vdash_{\ll} P : \phi_1 \longrightarrow \phi_2} \text{ (sub)}$$

$$\frac{}{\vdash_{\ll} x := a : \phi[x \mapsto a] \longrightarrow \phi} \text{ (prim)}$$

$$\frac{\vdash_{\ll} P_1 : \phi'' \longrightarrow \phi' \quad \vdash_{\ll} P_2 : \phi' \longrightarrow \phi}{\vdash_{\ll} P_1; P_2 : \phi'' \longrightarrow \phi} \text{ (seq)}$$

$$\frac{\vdash_{\ll} P_1 : \phi_{\mathrm{t}} \longrightarrow \phi \quad \vdash_{\ll} P_2 : \phi_{\mathrm{f}} \longrightarrow \phi}{\vdash_{\ll} \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : (b \Rightarrow \phi_{\mathrm{t}}) \wedge (\neg b \Rightarrow \phi_{\mathrm{f}}) \longrightarrow \phi} \text{ (if)}$$

$$\frac{\vdash_{\ll} P : \phi' \longrightarrow (b \Rightarrow \phi') \wedge (\neg b \Rightarrow \phi)}{\vdash_{\ll} \mathit{while}\ b\ \mathit{do}\ P : (b \Rightarrow \phi') \wedge (\neg b \Rightarrow \phi) \longrightarrow \phi} \text{ (while)}$$

Figure 10: "Type system" from standard weakest precondition calculus (= Hoare logic)

Here we replace all occurrences of d' by $d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^k$. The changes $[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]$ are idempotent hence this replacement of d' does not change $d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^k$. The following rule is therefore a special case of the last one:

$$\frac{\vdash_{\ll} P : d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^k \longrightarrow d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^k \quad d' \sqsubseteq d}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^k[\mathsf{pc} \mapsto d(\mathsf{pc})] \longrightarrow d}$$

On the other hand, the latter rule is a combination of the former and the rule (sub), hence the two last rules are equivalent. We now rename $d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^k$ to d', introducing some side conditions that $d'(x_i)$ must satisfy. This gives us the rule (while) in Fig. 9.

A.2 Hoare Logic

Hoare logic can be derived as a "type system" according to our method from the following backward analysis (essentially a weakest precondition calculus). We define $D = \mathcal{P}(\mathbf{Var} \to \mathbb{Z})$ (the set of all subsets of the state space, i.e., of all predicates on states) and $\sqsubseteq$ to be subset inclusion (implication). We define $[\![x := a]\!](\phi) = \phi[x \mapsto a]$, $[\![b]\!]_{\mathrm{t}}(\phi) = b \Rightarrow \phi$, $[\![b]\!]_{\mathrm{f}}(\phi) = \neg b \Rightarrow \phi$, $\mathsf{combine}(\phi, \phi') = \phi'$.

We get that $[\![P]\!]_{\ll}(\psi) = wp(\psi)$ where wp means the weakest precondition in the conventional sense. The corresponding "type system" is presented in Fig. 10, which is deductively equivalent to the standard formulation of Hoare logic.

It is also possible to derive a formulation of Hoare logic from a forward analysis (a strongest postcondition calculus). We refrain from presenting the details here.