A Simple Semantics and Static Analysis for Stack Inspection

Anindya Banerjee
IMDEA Software Institute*
Madrid, Spain
anindya.banerjee@imdea.org

David A. Naumann
Stevens Institute of Technology†
Hoboken, NJ 07030, USA
naumann@cs.stevens.edu

The Java virtual machine and the .NET common language runtime feature an access control mechanism specified operationally in terms of run-time stack inspection. We give a denotational semantics in “eager” form, and show that it is equivalent to the “lazy” semantics using stack inspection. We give a static analysis of safety, i.e., the absence of security errors, that is simpler than previous proposals. We identify several program transformations that can be used to remove run-time checks. We give complete, detailed proofs for safety of the analysis and for the transformations, exploiting compositionality of the eager semantics.

1 Introduction

System security depends in part on protecting resources through specified access control policies. For example, a policy may allow only some users the privilege to write the password file. A typical implementation of the policy found e.g., in UNIX operating systems, involves an access control list \( A \) which associates with each user name \( n \) their set of privileges \( A(n) \). When a program is running it has an associated user, normally the user who invoked the program. To write a file, a program for user \( n \) must make a system call, and that system code checks whether \( A(n) \) includes the privilege of writing to the file. In order for users to be able to change their passwords, the system code for this task executes in a special mode (“setuid” in UNIX); the effective user is the owner of the code (say, root) rather than the originator of the call (\( n \), which can write some files but not the password file).

The Java and .NET platforms offer a similar but more general security system [12, 13]. Instead of code being owned by a user or by “the system”, there can be code from a number of sources, called principals, which can be offered varying degrees of trust. Moreover, instead of associating a principal with a loadable executable file, principals can be associated with fragments such as class declarations. Another refinement is that privileges must be explicitly enabled, by an operation called doPrivileged. The intent is that a program only enables its privileges when they are needed; this “principle of least privilege” [12] may help isolate the effect of security bugs and may facilitate static analysis. Before executing a dangerous operation, a check is made that the associated privilege has been enabled and is authorized for the current principal. This check is specified in terms of an implementation called stack inspection. Each stack frame is marked with the principal associated with the code for that frame, and the frame also records the privileges that have been enabled. This is used by procedure checkPermission which inspects the current stack.

The above description of the security system is an operational one. While the mechanism itself is easy to understand, it may over-constrain implementations, and it is difficult to analyze. Analysis is of interest, e.g., to determine whether a program can exhibit security exceptions when given its expected permissions.
permissions. Implementations of procedure calls do not always push stack, e.g., owing to inlining or tail call optimization. To understand the security properties achieved, and to optimize performance, we need analyses that capture the security model more abstractly.

Our contribution is threefold: (i) We give a denotational semantics in “eager” form, and show that it is equivalent to the “lazy” semantics using stack inspection. (ii) We give a static analysis of safety, i.e., the absence of security errors. (iii) We identify several program transformations including some that can be used to remove run-time checks. We give detailed proofs for the analysis and for the transformations, exploiting compositionality of the eager semantics and simplicity of program equivalence in denotational semantics.

Related work. Skalka and Smith [17] give an operational semantics and use it to justify a static analysis of safety specified by a type system. Their type system is complicated by the choice of using a constraint system which is the basis for a type inference algorithm. We use a similar type system, but prefer to separate the specification of an analysis from algorithms to perform the analysis. We also include recursion in the language. Their semantics is easily seen to model the operational descriptions of stack inspection, but it has the usual shortcomings of operational semantics; for example, proofs ultimately go by induction on computations.

Wallach, Appel and Felten [19] model the mechanism with an operational semantics that manipulates formulas in a logic of authentication [1]. They show that the particular logical deductions corresponding to checkPermission can be decided efficiently, and propose an implementation called “security passing style” in which the security state is calculated in advance. The only result proven is equivalence of the two implementations. They do not include recursion or higher order functions, and the formal semantics is not made explicit. Although the use of logic sheds some light on the security properties achieved by the mechanism, the approach requires a considerable amount of theory that is not directly germane to analyzing safety or justifying optimizations.

Security passing style can be seen as a presentation of the eager means of evaluating security checks mentioned by Li Gong [12]. The eager semantics facilitates proofs, but JVM and CLR implementations use lazy semantics which appears to have better performance [12, 19, 13].

Pottier et al. [15] formalize the eager semantics by a translation into a lambda calculus, λsec, with operations on sets. Using an operational semantics for the calculus, a proof is sketched of equivalence with stack semantics. Using a very general framework for typing, a static analysis is given and a safety result is sketched. The language extends the language of Skalka and Smith [17] by adding permission tests. The works [17] and [15] aim to replace dynamic checks by static ones, but do not consider program transformations as such.

This paper originated as a technical report more than a decade ago [3]. At about the same time, and independently, Fournet and Gordon [11] investigated an untyped variant of λsec. They develop an equational theory that can be used to prove the correctness of code optimizations in the presence of stack inspection. A prime motivation for their work was the folklore that well-known program optimizations such as inlining and tail call elimination are invalidated by stack inspection. Their technical development uses small-step operational semantics and contextual equivalence of programs. To prove an extensive collection of program equivalences they develop a form of applicative bisimilarity. Additionally, they prove the equivalence of the lazy and eager implementations of stack inspection. This equivalence is also proved in Skalka’s Ph.D. dissertation [16 Theorem 4.1, Chapter 4], where lazy and eager are termed “backward” and “forward” stack inspection respectively.

Where Fournet and Gordon point out how tail call elimination can be invalidated by stack inspection,
Clements and Felleisen [10] consider program semantics at the level of an abstract machine, namely the CESK machine. With this semantics they are able to show tail call optimization can be validated in full generality, and with its expected space savings. This is explored further by Ager et al [2]. They inter-derive a reduction semantics for the untyped variant of the $\lambda_{sec}$-calculus, an abstract machine, and a natural semantics, both without and with tail-call optimization. By unzipping the context in the abstract machine, they connect these semantics to Wallach et al.’s security passing style, characterize stack inspection as a computational monad, and combine this monad with the monad of exceptions.

We treat a simply typed language similar to $\lambda_{sec}$, but with recursion. In contrast to the cited works we use a denotational semantics, which is straightforward; in fact, once the meanings of types are specified, the rest of the specification (i.e., meanings of expressions) follows easily. The simplicity of our model makes it possible to give a self-contained formal semantics and succinct but complete formal proofs. For example, denotational equality is a congruence simply because the semantics is compositional. We have not formally connected the semantics with an operational one. Adequacy seems obvious. Full abstraction is not obvious, but we have proved many of the contextual equivalences of Fournet and Gordon [11] and expect the remainder to be straightforward to show.

In addition to considering program transformations, Fournet and Gordon [11] address the question of what security properties can be enforced by stack inspection. They consider a variation that tracks history in the sense of what code has influenced the result of a computation. Pistoia et al [14] propose a variation that tracks implicit influences as well. The authors [5] propose another combination of information flow tracking with stack inspection, using a type and effect system where security types for functions are dependent on available permissions. In the interim, other code-based access mechanisms have been introduced (e.g., static permissions in the Android platform) and there have been further development of static analyses for security properties based on linear temporal logic [7, 6, 8, 18], but there seems to be little additional work on program equivalence in the presence of stack inspection.

**Outline.** The next section explains stack inspection informally, and it introduces our language. Section 3 gives the eager denotational semantics. Section 4 gives the static analysis for safety, including examples and correctness proof. Section 5 proves a number of representative program transformations. Section 6 shows how all checks can be removed from safe programs. Section 7 gives the stack-based denotational semantics and shows that it is equivalent to the eager semantics. Section 8 concludes.

## 2 Overview and language

Each declared procedure is associated with a principal $n$. We call $n$ the signer, and write $\text{signs } n e$ for a signed expression, because typically $n$ is given by a cryptographic signature on a downloaded class file. During execution, each stack frame is labeled with the principal that signs the function, as well as the set $P$ of privileges that have been explicitly enabled during execution of the function. For our purposes, a frame is a pair $\langle n, P \rangle$, and a nonempty stack is a list $\langle n, P \rangle :: S$ with $\langle n, P \rangle$ the top. There should be an initial stack $S_0 = \langle n_0, \emptyset \rangle :: \text{nil}$ for some designated $n_0$. An expression is evaluated in a stack $S$ and with an environment $h$ that provides values for its free variables.

Java provides operations to enable and disable a privilege, i.e., to add it to the stack frame or remove it. Normally these are used in bracketed fashion, as provided by procedure $\text{doPrivileged}$ which is given a privilege $p$ and an expression $e$ to evaluate. It enables $p$, evaluates $e$, and then disables $p$. Our

---

1 Thanks to Olivier Danvy for communicating this explanation.

2 Adding state is straightforward [4], but here we follow the cited works and confine attention to applicative expressions.
construct is written dopriv $p$ in $e$. The effect of dopriv $p$ in $e$ in stack $(n,P) :: S$ is to evaluate $e$ in stack $(n,P \cup \{p\}) :: S$, that is, to assert $p$ in the current frame. This is done regardless of whether $p$ is authorized for $n$.

Java’s checkPermission operation checks whether a certain privilege has been enabled and is authorized for the current principal. Checking is done by inspecting the current stack. Each dangerous code fragment should be guarded by a check for an associated privilege, so that the code cannot be executed unless the check has succeeded. (This can be assured by inspection of the code, or by other forms of analysis [9] but is beyond the scope of our paper.) In our syntax, a guarded expression is written check $p$ for $e$. The execution of an expression checked for privilege $p$ is to raise a security error, which we denote by $\star$, unless the following predicate is true of $p$ and the current stack.

$$
\text{chk}(p, \text{nil}) \iff \text{false}
$$

$$
\text{chk}(p,((n,P) :: S)) \iff p \in \mathcal{A}(n) \land (p \in P \lor \text{chk}(p,S))
$$

That is, a privilege is enabled for a stack provided there is some frame $(n,P)$ with $p \in P$ and $p$ is authorized for $n$ and is authorized for all principals in frames below this one.

A direct implementation in these terms requires inspecting some or all of the stack frames. The implementation is lazy in that no checking is performed when a privilege is enabled, only when it is needed to actually perform a guarded operation. On the other hand, each check incurs a significant cost, and in secure code the checks will never fail. Static analysis can detect unnecessary checks, and justify security-preserving transformations.

A stack $S$ determines a set $\text{privs} S$ of enabled, authorized privileges, to wit:

$$
p \in \text{privs} S \iff \text{chk}(p,S)
$$

This gives rise to a simple form of eager semantics: instead of evaluating an expression in the context of a stack $S$, we use $\text{privs} S$, along with the current principal which appears on top of $S$. The eager, stack-free semantics is given in Section 3 and we will use this semantics exclusively in the static analysis and program transformations that follow in Sections 4 and 5.

A denotational semantics that uses explicit stacks will be deferred until Fig. 4 of Section 7. As mentioned previously, we will then take up the equivalence of the two semantics.

The language constructs are strict in $\star$: if a subexpression raises a security error, so does the entire expression. In Java, security errors are exceptions that can be caught. Thus it is possible for a program to determine whether a checkPermission operation will succeed. Rather than model the full exception mechanism, we include a construct test $p$ then $e_1$ else $e_2$ which evaluates $e_1$ if $
\text{chk}(p,S)$ succeeds in the current stack $S$, and evaluates $e_2$ otherwise. Note that security error $\star$ is raised only by the check construct, not by test or dopriv.

In Java, the call of a procedure of a class signed by, or otherwise associated with, $n$, results in a new stack frame for the method, marked as owned by $n$. We model methods as function abstractions, but whereas Skalka and Smith use signed abstractions, we include a separate construct signs $n\:e$.

Evaluation of signs $n\:e$ in stack $S$ goes by evaluating $e$ in stack $(\langle n,\emptyset \rangle :: S)$. For example, given a stack $S$, the evaluation of the application

3Fournet and Gordon [11] also use a freely applicable construct for signing. Moreover they identify principals with sets of permissions: their “framed” expression $R[e]$ is like our signs $n\:e$ for $n$ with $\mathcal{A}(n) = R$. 

$$
(\text{fun } x. \text{signs user writepass}(x))"\text{myName}"
$$
amounts to evaluating \texttt{writepass(“myName”) in the stack \langle user, \emptyset \rangle :: S}.

We separate \texttt{signs} from abstractions because it helps disentangle definitions and proofs, e.g., these constructs are treated independently in our safety result. On the other hand, unsigned abstractions do not model the Java mechanism. In our consistency result, Theorem 7.2, we show that our semantics is equivalent to stack inspection for all standard expressions, i.e., those in which the body of every abstraction is signed.

### 2.1 Syntax and typing

Given are sets Principals and Privileges, and a fixed access control list \( \mathcal{A} \) that maps Principals to sets of privileges. In the grammar for data types and expressions, \( n \) ranges over Principals and \( p \) over Privileges. Application associates to the left. We include recursive definitions for expressiveness, and simple abstractions \texttt{fun x.e} which, while expressible using \texttt{letrec}, are easier to understand in definitions and proofs. For simplicity, the only primitive type is \texttt{bool}, and the only type constructor is for functions. Products, sums, and other primitive types can be added without difficulty. Throughout the paper we use \texttt{true} to exemplify the treatment of constants in general.

\[
\begin{align*}
t & ::= \texttt{bool} \mid (t \to t) \\
e & ::= \texttt{true} \mid x \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\
& \quad \mid \texttt{fun } x. e \mid \texttt{letrec } f(x) = e_1 \texttt{ in } e_2 \\
& \quad \mid \texttt{signs } n \ e \mid \texttt{dopriv } p \ \texttt{in } e \mid \texttt{check } p \ \texttt{for } e \mid \texttt{test } p \ \texttt{then } e_1 \texttt{ else } e_2
\end{align*}
\]

A signed abstraction \( ^n \lambda x. e \) in the language of Skalka and Smith is written \texttt{fun x. signs } n \ e \texttt{ in ours. Our safety result can be proved without restriction to expressions of this form. But for the eager semantics to be equivalent to stack semantics, it is crucial that function bodies be signed so the semantics correctly tracks principals on behalf of which the body of an abstraction is evaluated.}

**Definition 2.1 (Standard expression)** An expression is standard if for every subexpression \texttt{fun x.e} or \texttt{letrec }\( f(x) = e_1 \texttt{ in } e_2 \) we have that \( e \) is \texttt{signs } n \ e' for some \( n, e' \).

Well-formed expressions are characterized by typing judgements \( D \vdash e : t \) which express that \( e \) has type \( t \) where free identifiers are declared by \( D \). A typing context \( D \) is a labeled tuple of declarations \( \{ x_1 : t_1, \ldots, x_k : t_k \} \). We write \( D, x : t \) for the extended context \( \{ x_1 : t_1, \ldots, x_k : t_k, x : t \} \), and \( D, x_i \) for the type of \( x_i \). The typing rules are given in Figure 1.

### 2.2 The password example

As an example of the intended usage, we consider the problem of protecting the password file, using a privilege \( p \) for changing password and \( w \) for writing to the password file. The user is authorized to change passwords: \( \mathcal{A}(\text{user}) = \{ p \} \). Root is authorized to change passwords and to write the password file: \( \mathcal{A}(\text{root}) = \{ p, w \} \). Suppose \texttt{hwWrite} is the operating system call which needs to be protected from direct user access. The system provides the following code, which guards \texttt{hwWrite} with the privilege \( w \).

\[
\begin{align*}
\texttt{writepass} &= \texttt{fun x. signs } \texttt{root check } w \texttt{ for } \texttt{hwWrite}(x, ”/etc/password”) \\
\texttt{passwd} &= \texttt{fun x. signs } \texttt{root check } p \texttt{ for } \texttt{dopriv } w \texttt{ in } \texttt{writepass}(x)
\end{align*}
\]
Consider the following user programs.

\[
\begin{align*}
\text{bad1} &= \text{signs user writepass}("mypass") \\
\text{bad2} &= \text{signs user dopriv w in writepass}("mypass") \\
\text{use} &= \text{signs user dopriv p in passwd}("mypass")
\end{align*}
\]

Here \text{bad1} raises a security exception because \text{writepass} checks for privilege \text{w} which is not possessed by \text{user}. The user can try to enable \text{w}, as in \text{bad2}, but because \text{w} is not authorized for \text{user} the exception is still raised. By contrast, \text{use} does not raise an exception: function \text{passwd} checks for privilege \text{p} which is possessed by \text{user}, and it enables the privilege \text{w} needed by \text{writepass}. Using transformations discussed in Section 5, checks that never fail can be eliminated. For example, the analysis will show that \text{use} is safe, and the transformations will reduce \text{use} to

\[
\text{signs user signs root hwWrite}("mypass","/etc/password")
\]

## 3 Denotational semantics

This section gives the eager denotational semantics.

### 3.1 Meanings of types and type contexts

A cpo is a partially ordered set with least upper bounds of ascending chains; it need not have a least element. Below we define, for each type \( t \), a cpo \( \llbracket t \rrbracket \). We assume that \( \bot \) and \( \ast \) are two values not in \{true, false\} and not functions; this will ensure that \( \{\bot, \ast\} \cap \llbracket t \rrbracket = \emptyset \) for all \( t \). We will identify \( \bot \) with non-termination and \( \ast \) with security errors. For cpo \( C \), define \( C_{\bot,\ast} = C \cup \{\bot, \ast\} \), ordered as the disjoint union of \( C \) with \{\ast\}, lifted with \( \bot \). That is, for any \( u, v \in C_{\bot,\ast} \), define \( u \leq v \) iff \( u = \bot \), \( u = v \), or \( u \) and \( v \) are in \( C \) and \( u \leq v \) in \( C \).

We define \( \llbracket \text{bool} \rrbracket = \{\text{true, false}\} \), ordered by equality. We also take the powerset \( \mathcal{P}(\text{Privileges}) \) to be a cpo ordered by equality. Define

\[
\llbracket t_1 \to t_2 \rrbracket = \mathcal{P}(\text{Privileges}) \to \llbracket t_1 \rrbracket \to \llbracket t_2 \rrbracket_{\bot,\ast}
\]
where \( \to \) associates to the right and denotes continuous function space, ordered pointwise. Note that lubs are given pointwise. Also, \( \llbracket t_1 \to t_2 \rrbracket \) does not contain \( \bot \) but it does have a least element, namely the constant function \( \lambda P. \lambda d. \bot \).

Principals behave in a lexically scoped way. By contrast, privileges are dynamic and vary during execution; this is reflected in the semantics of the function type.

Let \( D = \{ x_1 : t_1, \ldots, x_k : t_k \} \) be a type context. Then \( \llbracket D \rrbracket \) is defined to be the set \( \{ x_1 : \llbracket t_1 \rrbracket, \ldots, x_k : \llbracket t_k \rrbracket \} \) of labeled tuples of appropriate type. If \( h \) is such a record, we write \( h.x_i \) for the value of field \( x_i \).

If \( D \) is the empty type context \( \emptyset \), then the only element of \( \llbracket D \rrbracket \) is the empty record \( \{ \} \). For \( h \in \llbracket D \rrbracket \) and \( d \in \llbracket t \rrbracket \) we write \( [h | x \mapsto d] \) for the extended record in \( \llbracket D, x : t \rrbracket \).

### 3.2 Meanings of expressions

An expression judgement denotes a function

\[
\llbracket D \vdash e : t \rrbracket \in \text{Principals} \to \mathcal{P}(\text{Privileges}) \to \llbracket D \rrbracket \to \llbracket t \rrbracket \bot^*.
\]

Given a principal \( n \), a set \( P \in \mathcal{P}(\text{Privileges}) \) denoting privileges required by \( e \), and environment \( h \in \llbracket D \rrbracket \), the meaning of \( \llbracket D \vdash e : t \rrbracket | nP \) is either \( \bot \) or \( * \) or an element of \( \llbracket t \rrbracket \).

In contrast with the work of Fournet and Gordon we do not restrict \( P \) to be a subset of \( \mathcal{A}(n) \), though it can easily be done—simply by giving the denotation this dependent type:

\[
\llbracket D \vdash e : t \rrbracket \in (n : \text{Principals}) \to \mathcal{P}(\mathcal{A}(n)) \to \llbracket D \rrbracket \to \llbracket t \rrbracket \bot^*.
\]  \hspace{1cm} (1)

In programs of interest, signed at the top level, most expressions will in fact be applied to permission sets that satisfy the restriction. Later we observe that the restriction is need for validity of some transformations, but surprisingly few of them.

In the denotational semantics (Figure 2), we use the metalanguage construct, \texttt{let d = E1 in E2}, with the following semantics: if the value of \( E_1 \) is either \( \bot \) or \( * \) then that is the value of the entire let expression; otherwise, its value is the value of \( E_2 \) with \( d \) bound to the value of \( E_1 \). The semantics of if-then-else is \(*\)-strict in the guard. We also write \( P \cup_n \{ p \} \) for \( \text{if } p \in \mathcal{A}(n) \text{ then } P \cup \{ p \} \text{ else } P \).

The semantics is standard for the most part. We will only explain the meanings of the expressions that directly concern security. In what follows, we will assume, unless otherwise stated, that expression \( e \) is signed by principal \( n \) and is computed with privilege set \( P \) and in environment \( h \).

The meaning of \( \text{signs } n' e \) is the meaning of \( e \), signed by \( n' \), computed with privilege set \( P \cap \mathcal{A}(n') \), in \( h \). To illustrate the idea, consider Li Gong’s example [12, Section 3.11.2]. A game applet, \texttt{applet}, has a method that calls \texttt{FileInputStream} to open the file containing the ten current high scores. In our semantics, this scenario entails finding the meaning of \( \text{signs } \text{system } \text{FileInputStream} \), as invoked under some privilege set \( P \subseteq \mathcal{A}(\text{applet}) \); and, this means we need to find the meaning of \( \text{FileInputStream} \) (i.e., whether read privileges are enabled) under the privilege set \( P \cap \mathcal{A}(\text{system}) \). Assuming \texttt{system} has all privileges, this reduces to checking if \( \text{applet} \) has been granted permission to read. If it has not been granted the permission, the file will not be read, even though it calls system code to do so.

The meaning of \( \text{dopriv } p \text{ in } e \) is the meaning of \( e \) computed with privilege set \( P \cup \{ p \} \) if \( p \in \mathcal{A}(n) \), and is the meaning of \( e \) computed with privilege set \( P \) if \( p \notin \mathcal{A}(n) \). The meaning of \( \text{check } p \text{ for } e \) is a security error if \( p \notin P \); otherwise, the meaning is that of \( e \). Finally, the meaning of \( \text{test } p \text{ then } e_1 \text{ else } e_2 \) is the meaning of \( e_1 \) or \( e_2 \) according as \( p \in P \) or \( p \notin P \).

We leave it to the reader to check that the semantics of each construct is a continuous function of the semantics of its constituent expressions, so the semantics of recursion is well defined.
4 Static Analysis

The denotational semantics in Section 3 gives a dynamic or run-time view of safety; if a program is safe, its execution will not yield *. In this section, we specify a type system that statically guarantees safety; if a program is well-typed in the system then it is safe. One may utilize the static analysis for optimizing programs e.g., removing redundant checks of privileges at run-time.

The static analysis is specified by an extended form of typing judgement. The idea is to give not only the type of an expression, but a principal \( n \) and set \( P \) of privileges for which the expression is safe. An arrow type \( t_1 \rightarrow t_2 \) denotes functions dependent on a set of privileges, and the static analysis uses annotated types to track sets of privileges adequate for safety. We adopt a Greek notational style for types in the static analysis. Letting \( \Pi \) range over sets of privileges, annotated types, \( \theta \), are defined by

\[
\theta ::= \text{bool} | (\theta_1 \xrightarrow{n} \theta_2)
\]

For this syntax to be finitary, one could restrict \( \Pi \) to finite sets, but we have no need for such restriction in our proofs. An expression typed \( \theta_1 \xrightarrow{n} \theta_2 \) signifies that its application may require at least the privileges \( \Pi \) for safe execution.

4.1 Type-based analysis

The analysis is specified by the typing judgement \( \Delta; n \vdash e : \theta, \Pi \). In words, expression \( e \) signed by principal \( n \) and typed in context \( \Delta \), has (annotated) type \( \theta \) and is safe provided at least the set \( \Pi \) of privileges are enabled. Figure 3 gives the specification.

Constant \texttt{true}, identifiers, and anonymous functions of the form \texttt{fun} \( x \). \( e \) are all safe: they do not require any privileges be enabled for safe execution. However, the body \( e \) in \texttt{fun} \( x \). \( e \), may require a set
of privileges \( \Pi \) be enabled. This is manifest in the type \( e_1 \xrightarrow{n} e_2 \). The latent privileges, \( \Pi \), get exposed during an application, \( e_1 e_2 \). Say \( e_1 \) has type \( \theta_1 \xrightarrow{n} \theta_2 \); if \( \Pi_1 \) may be enabled during \( e_1 \)'s execution, and \( \Pi_2 \) may be enabled during \( e_2 \)'s execution, then application itself may require \( \Pi \) be enabled; hence \( \Pi \cup \Pi_1 \cup \Pi_2 \) may be enabled during the execution of \( e_1 e_2 \). The application rule also uses subtyping, as discussed in the sequel.

The analysis for \( \text{check} \ p \ \text{for} \ e \) requires that in addition to privileges enabled for \( e \), the privilege \( p \) be enabled so that the check is safe. If \( \Pi \) is the set of privileges that may be enabled during the execution of \( \text{dopriv} \ p \ \text{in} \ e \), then \( p \) can be assumed to be enabled during the execution of \( e \), provided \( p \in \mathcal{A}(n) \).

Finally, for \( \text{signs} \ n' \ e \) the only privileges that should be enabled are the ones authorized for \( n' \). Note that a signed expression can occur in a term with a different owner, so it is not the case that \( \Pi \subseteq \mathcal{A}(n) \) for every derivable \( \Delta; n \vdash e : \theta, \Pi \).

### 4.2 Subtyping

Where Skalka and Smith [17] use a constraint-based type system whose constraints subsequently must be solved, our analysis is syntax-directed. In some sense, our system gives minimal types and privilege assumptions. (Pottier et al.'s system [15] enjoys a principal types property also. Reasoning about minimal security contexts for code invocation is also considered in several papers by Besson et al. [7, 6, 8].) We do not formalize this notion, but informally it sheds light on the specification of the analysis. In the case of values, such as variables and abstraction, the privilege set is empty. In the case of \( \text{check} \ p \ \text{for} \ e \), the rule adds the checked privilege \( p \) to the “minimal” privileges of \( e \), and similarly for the other security constructs. In the case of conditional, a union is formed from the “minimal” privileges of the constituent expressions, and the types of the constituents are the same as the type of the conditional. By contrast, in the case of application \( e_1 e_2 \), the “minimal” types and privileges for \( e_1 \) and \( e_2 \) need not match exactly. So we define a relation of subtyping with the informal meaning that \( \theta' \leq \theta \) provided the privileges required

---

4Pottier et al. [15] use unification of row variables, in a relatively complicated system.
by \(\theta'\) are contained in those required by \(\theta\). This is significant only in case \(e_2\) has functional type, in which case the latent privileges of \(e_2\) should be among those of \(e_1\).

Subtyping is defined as the least relation \(\leq\) with \(\text{bool} \leq \text{bool}\) and, for arrow types, \(\theta_1 \xrightarrow{\Pi_1} \theta_1' \leq \theta_2 \xrightarrow{\Pi_2} \theta_2'\) provided \(\theta_1 \leq \theta_2\), \(\theta_1' \subseteq \theta_2'\), and \(\Pi_1 \subseteq \Pi_2\).

To relate the semantics to the static analysis, we need the ordinary type \(\theta^*\) obtained by erasing annotations. This is defined by induction on \(\theta\), to wit: \(\text{bool}^* = \text{bool}\) and \((\theta_1 \xrightarrow{\Pi_1} \theta_2)^* = \theta_1^* \rightarrow \theta_2^*\). It is easy to show that if \(\theta_1 \leq \theta_2\), then \(\theta_1^* = \theta_2^*\).

Due to subtyping, an expression can have more than one annotated type and satisfy more than one judgement. But a derivable judgement \(\Delta; n \vdash e : \theta\), \(\Pi\) has only one derivation, which is dictated by the structure of \(e\). Proofs in the sequel will go by “induction on \(e\)”, meaning induction on the derivation of some judgement \(\Delta; n \vdash e : \theta\), \(\Pi\).

### 4.3 Examples

For any \(n\), the expressions in the password example (Section 2.2) can be analyzed as follows.

\[
\emptyset; n \vdash \text{writepass} : \text{string} \xrightarrow{\text{w}} \text{void}, \emptyset \\
\emptyset; n \vdash \text{passwd} : \text{string} \xrightarrow{\text{p}} \text{void}, \emptyset \\
\emptyset; n \vdash \text{use} : \text{void}, \emptyset
\]

This confirms that \(\text{use}\) is safe. On the other hand, there is no \(\Pi\) such that \(\emptyset; n \vdash \text{bad1 : void}, \Pi\) or \(\emptyset; n \vdash \text{bad2 : void}, \Pi\). Such \(\Pi\) must satisfy \(w \in \Pi\) for the application of \(\text{writepass}\), owing to the rules for application and for \(\text{dopriv in}\). And \(\Pi\) must satisfy \(\mathcal{A}(\text{user}) \subseteq \Pi\) by the rule for signs.

Here is another example, inspired by ones in Skalka and Smith [17]. Define the following standard expressions:

\[
lp = \text{fun } f. \text{signs } n (\text{fun } x. \text{signs } n (\text{dopriv } p \text{ in } (f x))) \\
cp = \text{fun } x. \text{signs } n (\text{check } p \text{ for } x)
\]

The reader can verify that one analysis for \(cp\) is given by the typing \(\Delta; n \vdash cp : \text{bool} \xrightarrow{\text{p}} \text{bool}, \emptyset\) and that the typing demands \(p \in \mathcal{A}(n)\). Similarly, the reader can verify that one possible analysis for \(lp\) is given by the typing \(\Delta; n \vdash lp : \text{bool} \xrightarrow{\text{p}} \text{bool} \xrightarrow{\text{a}} \text{bool} \xrightarrow{\text{a}} \text{bool}, \emptyset\).

For all \(P \in \mathcal{P}(\text{Privileges})\), for all \(h : \Delta^*\), we can show (omitting types and some steps),

\[
\llbracket lp \rrbracket nPh = \lambda P_1. \lambda d_1. \lambda P_2. \lambda d_2. \llbracket \text{dopriv } p \text{ in } f x \rrbracket n(P_2 \cap \mathcal{A}(n))[h \mid f \mapsto d_1, x \mapsto d_2]
\]

\[
\llbracket cp \rrbracket nPh = \lambda P'_1. \lambda d'_1. \llbracket \text{if } p \in (P'_1 \cap \mathcal{A}(n)) \text{ then } \llbracket x \rrbracket n(P'_1 \cap \mathcal{A}(n))[h \mid x \mapsto d'_1] \text{ else } \star
\]

Let \(F = \llbracket lp \rrbracket nPh\), let \(d = \llbracket cp \rrbracket nPh\), and let \(G = \llbracket (lp \ cp) \rrbracket nPh\). Then

\[
\llbracket (lp \ cp) \rrbracket nPh = FPd
\]

\[
= \lambda P_2. \lambda d_2. \text{if } p \in ((P_3 \cup \{p\}) \cap \mathcal{A}(n)) \text{ then } d_2 \text{ else } \star
\]

\[
= \lambda P_2. \lambda d_2. \text{ because } p \in \mathcal{A}(n)
\]

\[
\llbracket (lp \ cp) \text{true} \rrbracket nPh = GP(\llbracket \text{true} \rrbracket nPh)
\]

\[
= \text{true}
\]

Hence \((lp \ cp)\text{true}\) is safe in any environment and typable as \(\Delta; n \vdash (lp \ cp)\text{true} : \text{bool}, \emptyset\).
4.4 Safety of the analysis

**Theorem 4.1 (Safety)** Suppose \( \varnothing; n \vdash e : \theta \), \( \Pi \) is derivable. Then for all \( P \in \mathcal{P}(\text{Privileges}) \) with \( \Pi \subseteq P \), it is the case that \( \llbracket \varnothing \vdash e : \theta^* \rrbracket \#P\{\} \neq * \).

**Proof:** Immediate consequence of Lemma 4.5 below. ■

In order to serve as an adequate induction hypothesis, the lemma strengthens the theorem by allowing judgements with non-empty contexts. But this is not enough. Values at arrow types are functions that be safe with respect to the privilege set \( \Pi \) annotating their type.

**Definition 4.2** For each annotated type \( \theta \) the predicate \( \text{safe} \) on \( \llbracket \theta^* \rrbracket_{\downarrow} \) is defined as follows:

\[
\text{safe} \theta(\bot) \iff \text{true} \quad \text{and} \quad \text{safe} \theta(*) \iff \text{false} \quad \text{for all} \quad \theta.
\]

For values other than \( \bot \) and \( * \), the definition is by induction on structure of \( \theta \).

\[
\begin{align*}
\text{safe} \text{ bool}(b) & \iff \text{true} \\
\text{safe} (\theta_1 \rightarrow \theta_2)(f) & \iff \forall P \in \mathcal{P}(\text{Privileges}). \forall d \in \llbracket \theta_1^* \rrbracket. \\
\Pi \subseteq P \land \text{safe} \theta_1(d) & \implies \text{safe} \theta_2(fPd)
\end{align*}
\]

The predicate \( \Delta \) on \( \llbracket \Delta^* \rrbracket \) is defined by \( \Delta(h) \iff \forall x \in \text{dom}(h). \text{safe}(\Delta.x)(h,x) \). Recall that \( h.x \neq \bot \) and \( h.x \neq * \), because \( \bot \notin \llbracket t \rrbracket \) and \( * \notin \llbracket t \rrbracket \), for all \( t \).

**Fact 4.3** \( \theta \leq \theta' \) and \( \text{safe} \theta \, d \) imply \( \text{safe} \theta' \, d \).

**Proof:** By induction on derivation of \( \theta \leq \theta' \). The result is clear for \( \text{bool} \leq \text{bool} \). For \( (\theta_1 \rightarrow \theta_2) \leq (\theta'_1 \rightarrow \theta'_2) \), assume \( \text{safe} (\theta_1 \rightarrow \theta_2) \, f \). To show \( \text{safe} (\theta'_1 \rightarrow \theta'_2) \, f \), consider any \( P \in \mathcal{P}(\text{Privileges}) \), such that \( \Pi' \subseteq P \), and any \( d \in \llbracket \theta_1^* \rrbracket \) with \( \text{safe} \theta'_1 \, d \). From the subtyping, we know that \( \Pi \subseteq \Pi' \), hence \( \Pi \subseteq P \). Moreover, by induction on derivation of \( \theta'_1 \leq \theta_1 \), we obtain \( \text{safe} \theta'_1 \, d \) implies \( \text{safe} \theta_1 \, d \). Hence from assumption \( \text{safe} (\theta_1 \rightarrow \theta_2) \, f \), we obtain \( \text{safe} \theta_2(fPd) \) holds. Now by induction on derivation \( \theta_2 \leq \theta'_2 \), we obtain \( \text{safe} \theta'_2(fPd) \). ■

**Lemma 4.4** The predicate \( \text{safe} \) preserves lubs. That is, for any \( \theta \), let \( u : \mathbb{N} \rightarrow \llbracket \theta^* \rrbracket_{\downarrow} \) be an ascending chain. Then, \( \forall i. \text{safe} \theta (u_i) \) implies \( \text{safe} \theta (\llbracket i \rrbracket, u_i) \).

**Proof:** By structural induction on \( \theta \). When \( \theta = \text{bool} \), the assumption \( \text{safe} \theta (u_i) \) implies \( u_i \neq * \) for each \( i \), so \( \llbracket i \rrbracket, u_i \) is true or false or \( \bot \). Thus the result holds by definition of \( \text{safe} \).

When \( \theta = (\theta_1 \rightarrow \theta_2) \), assume \( P \in \mathcal{P}(\text{Privileges}) \) and \( d \in \llbracket \theta_1^* \rrbracket \), such that \( \Pi \subseteq P \) and \( \text{safe} \theta_1(d) \). Then, from assumption \( \text{safe} (\theta_1 \rightarrow \theta_2) \, u_i \) we obtain \( \text{safe} \theta_2(u_i \, Pd) \) holds for every \( i \). By the induction hypothesis on \( \theta_2 \), we get \( \text{safe} \theta_2 (\llbracket i \rrbracket (u_i \, Pd)) \). Lubs are pointwise, so we get \( \text{safe} \theta_2 (\llbracket i \rrbracket (u_i \, Pd)) \) by the induction hypothesis on \( \Pi \subseteq P \). ■

**Lemma 4.5** Suppose \( \Delta; n \vdash e : \theta \), \( \Pi \) is derivable. Then for all \( P \in \mathcal{P}(\text{Privileges}) \), for all \( h \in \llbracket \Delta^* \rrbracket \), if \( \text{safe} \Delta(h) \) and \( \Pi \subseteq P \) then \( \text{safe} \theta (\llbracket \Delta^* \vdash e : \theta^* \rrbracket \#P\{\}) \).

Theorem 4.1 follows from the lemma because \( \text{safe} \varnothing\{\} \) and \( \text{safe} \theta (\llbracket \varnothing \vdash e : \theta^* \rrbracket \#P\{\}) \) implies \( \llbracket \varnothing \vdash e : \theta^* \rrbracket \#P\{\} \neq * \).

Another consequence of the lemma is that the language admits additional constants at all types, declared in an initial context \( D_0 \), provided the corresponding initial environment assigns a safe meaning to each identifier in \( D_0 \).
Proof: of Lemma. Go by induction on the typing derivation, $\Delta; n \vdash e : \theta$, $\Pi$. Throughout, we assume $P \in \mathcal{P}(\text{Privileges})$ and $h \in \llbracket \Delta^* \rrbracket$ and safe $\Delta(h)$ and $\Pi \subseteq P$, and also let $u = \llbracket \Delta^* \vdash e : \theta^* \rrbracket nPh$ for each case of $e$.

- Case true: Then, $u = tru$ so safe $\mathit{bool}(u)$ by definition safe.
- Case $x$: Then, $u = h.x$ and safe $\theta(h.x)$ follows, by definition safe, from the assumption safe $\Delta(h)$.
- Case if $e$ then $e_1$ else $e_2$: Then $\Pi_1 \cup \Pi_2 \cup \Pi_3 \subseteq P$, and
  \[
  u = \mathit{if}~b~\mathit{then}~\llbracket \Delta^* \vdash e_1 : \theta^* \rrbracket nPh~\mathit{else}~\llbracket \Delta^* \vdash e_2 : \theta^* \rrbracket nPh
  \]
  where $b = \llbracket \Delta^* \vdash \mathit{bool}(b) \rrbracket nPh$. By the induction hypothesis on the typing derivation of $e$, noting that $\Pi_1 \subseteq P$, we have safe $\mathit{bool}(b)$ and hence $b \neq \ast$. If $b = \perp$ then $u = \perp$ and $\perp$ is safe. Otherwise, $b = \mathit{true}$ or $b = \mathit{false}$. In the former case, by the induction hypothesis on the typing derivation of $e_1$, noting that $\Pi_2 \subseteq P$, we have safe $\theta(u)$. The case of $b = \mathit{false}$ is symmetric.
- Case fun $x$. $e$: Then $u = \lambda P'. \lambda d. \llbracket \Delta', x : \theta_1^* \vdash e : \theta_2^* \rrbracket nP'[h \mid x \mapsto d]$. Thus $u \neq \ast$. To prove safe $(\theta_1 \xrightarrow{n} \theta_2)(u)$, consider any $P'' \in \mathcal{P}(\text{Privileges})$ and any $d' \in \llbracket \theta_1^* \rrbracket$ such that $\Pi \subseteq P''$ and safe $\theta_1(d')$, to show safe $\theta_2(uP''d')$. By semantics, $uP''d' = \llbracket \Delta', x : \theta_1^* \vdash e : \theta_2^* \rrbracket nP'[h \mid x \mapsto d']$, so the induction hypothesis for $e$ yields safe $\theta_2(uP''d')$ provided that $\Pi \subseteq P''$ and safe $(\Delta, x : \theta_1)[h \mid x \mapsto d']$. We have $\Pi \subseteq P''$ by assumption, and safe $(\Delta, x : \theta_1)[h \mid x \mapsto d']$ follows from safe $\Delta(h)$ and safe $\theta_1(d')$.
- Case $e_1 \mathit{letrec} f(x) = e_1$ in $e_2$: Then, $\Pi_1 \cup \Pi_1 \subseteq P$.
  Now $u = \llbracket \Delta^* ; f : \theta_1^* \rightarrow \theta_2^* \rrbracket nP[h \mid f \mapsto \mathit{fix}~G], where
  \[
  G(g) = \lambda P'. \lambda d. \llbracket \Delta^*, f : \theta_1^* \rightarrow \theta_2^*, x : \theta_1^* \vdash e_1 : \theta_2^* \rrbracket nP'[h \mid f \mapsto g, x \mapsto d]\]
  To get safe $\theta(u)$ by induction for $e_2$, we need $\Pi_1 \subseteq P$ and
  
  safe $(\Delta, f : \theta_1 \xrightarrow{n} \theta_2)[h \mid f \mapsto \mathit{fix}~G]
  
  The former follows from the assumption $\Pi \cup \Pi_1 \subseteq P$. The latter follows from assumption, safe $\Delta(h)$, and safe $(\theta_1 \xrightarrow{n} \theta_2)(\mathit{fix}~G)$. We proceed to show safety of $\mathit{fix}~G$.
  Now $\mathit{fix}~G = \bigcup g_i$, where $g_0 = \lambda P''. \lambda d \in \llbracket \theta_1^* \rrbracket \perp$ and $g_{i+1} = G(g_i)$. And, safe $(\theta_1 \xrightarrow{n} \theta_2)(\mathit{fix}~G)$ is a consequence of the following claim:
  \[
  \forall i. \text{safe} (\theta_1 \xrightarrow{n} \theta_2)(g_i) \tag{2}
  \]
  Then from Lemma[4,4] we get safe $(\theta_1 \xrightarrow{n} \theta_2)(\bigcup g_i)$. It remains to show [2], for which we proceed by induction on $i$.
  Base case: Show safe $(\theta_1 \xrightarrow{n} \theta_2)(g_0)$. Assume any $P'' \in \mathcal{P}(\text{Privileges})$ and any $v \in \llbracket \theta_1^* \rrbracket$, such that $\Pi \subseteq P''$ and safe $\theta_1(v)$. Then $g_0P''v = \perp \neq \ast$ and safe $\theta_2(g_0P''v)$ holds.
We emphasize that this means extensional equality of the functions denoted by the two sides. In particular, optimization. This section shows how checks can be eliminated from the password example and also using the eager semantics it is straightforward to justify program transformations that can be used for

5 Some program transformations

Using the eager semantics it is straightforward to justify program transformations that can be used for optimization. This section shows how checks can be eliminated from the password example and also considers the proofs of some primitive equations from Fourment and Gordon’s work [11].

5.1 Transformations that eliminate checks

First, we list a series of program transformations that move checking of privileges “outwards”.

```plaintext
if e then check p for e1 else check p for e2 = check p for if e then e1 else e2
e1(check p for e2) = check p for e1 e2
test p then e1 else check p for e2 = check p for test p then e1 else e2
test p’ then check p for e1 else check p for e2 = check p for test p’ then e1 else e2
letrec f (x) = e1 in check p for e2 = check p for letrec f (x) = e1 in e2
check p for check p for e = check p for e
```

These are unconditional equalities, as the reader can verify using the denotational semantics (Figure 2). We emphasize that this means extensional equality of the functions denoted by the two sides. In particular, \( e = e' \) means \( \llbracket D \vdash e : t \rrbracket nPh = \llbracket D \vdash e' : t \rrbracket nPh \) for all \( n, P, h \). Corresponding to Fourment and
Definition 5.1 (p-purity) An expression e is p-pure if e has no sub-expressions of the form check p for e’ or test p then e’ else e”.

For each type t we define semantic p-purity as a predicate pure p t on \([t]\)\(_{\perp,*}\), as follows:

\[
\text{pure } p (t_1 \rightarrow t_2)(f) \iff \forall P \in \mathcal{P}(\text{Privileges}), \forall d \in \text{domain}(t_1).
\]

\[
\text{pure } p (t_1) \Rightarrow \text{pure } p (fPd) \land fPd = f(P - \{p\})d
\]

Finally, for environment \(h \in \text{domain}(D)\) we define pure p D(h) iff pure p t(h.x) for all x : t in D.

Lemma 5.2 Suppose \(u : \mathbb{N} \rightarrow \text{domain}(t_1 \rightarrow t_2)\) is an ascending chain. Then \(\forall i.\text{pure } p (t_1 \rightarrow t_2)(u_i)\) implies \(\text{pure } p (t_1 \rightarrow t_2)(\bigcup_i u_i)\).

Proof: By definition of pure and since joins are given pointwise. □

Lemma 5.3 If e is p-pure and typable as \(D \vdash e : t\), then for all n, P, h with pure p D(h) we have

\[
\text{pure } p t(\text{domain}(D) \vdash e : t)nPh = \text{domain}(D) \vdash e : t)(P - \{p\})h
\]

and pure p t(\text{domain}(D) \vdash e : t)nPh).

Proof: By induction on e. We observe for any D, n, P, h with pure p D (h)

- Case true: The equation is direct from the semantics, which is independent of P. For p-purity of true, the result holds by definition of pure p bool.
- Case x: the equation is direct from the semantics which is independent of P. For p-purity of \(\text{domain}(D) \vdash x : t\) nPh, the result holds by hypothesis on h.
- Case if e_1 then e_2 else e_3: straightforward use of induction.
- Case fun x. e: The equation holds because the semantics is independent of P. Purity holds by induction on e.
- Case e_1 e_2: To show the equation, we use that \(\text{domain}(D) \vdash e_1\) is p-pure, which holds by induction. To show purity, we again use purity of \(e_1\) as well as purity of \(e_2\).
- Case letrec f(x) = e_1 in e_2: By induction on \(e_2\), using Lemma 5.2.
- Case signs n' e: The equation is direct from semantics, using the fact that \((P \cap \mathcal{A}(n')) - \{p\} = (P - \{p\}) \cap \mathcal{A}(n')\).
Theorem 5.5
For all $\forall 298$

A Simple Semantics and Static Analysis for Stack Inspection

In the penultimate step, two uses are needed for the lemma: to remove it back.

Theorem 5.4
For all $\forall$

Proof: Let $\forall$

Case $\forall p \in e$: We first consider the case where $p'$ is distinct from $p$. We have

\[
[D \vdash \text{dopriv } p' \in e : t] n Ph
= [D \vdash e : t] n (P \sqcup_n \{p\}) h
= [D \vdash e : t] n ((P \sqcup_n \{p\}) \setminus \{p\}) h
= [D \vdash e : t] n (P \setminus \{p\} \sqcup_n \{p\}) h
= [D \vdash \text{dopriv } p' \in e : t] n (P \setminus \{p\}) h
\]

In case $p'$ is $p$ we have

\[
[D \vdash \text{dopriv } p \in e : t] n Ph
= [D \vdash e : t] n (P \sqcup_n \{p\}) h
= [D \vdash e : t] n ((P \setminus \{p\}) \sqcup_n \{p\}) h
= [D \vdash \text{dopriv } p \in e : t] n (P \setminus \{p\}) h
\]

The middle step is by cases on whether $p \in \mathcal{A}(n)$. If it is, the step holds by simply by definition of $\sqcup_n$. If not, the step holds by induction on $e$.

Case $\forall \check{p'}$ for $e$: Here $p'$ is distinct from $p$, by $p$-purity. We observe

\[
[D \vdash \check{p'} \text{ for } e : t] n Ph
= \text{ if } p' \in P \text{ then } [D \vdash e : t] n Ph \text{ else } \star
= \text{ if } p' \in P \setminus \{p\} \text{ then } [D \vdash e : t] n (P \setminus \{p\}) h \text{ else } \star
= \text{ if } p' \in P \setminus \{p\} \text{ then } [D \vdash \check{p'} \text{ for } e : t] n (P \setminus \{p\}) h
\]

Case $\forall \text{test} p' \text{ then } e_1 \text{ else } e_2$: Again, $p'$ is distinct from $p$, and the argument is similar to check.

Theorem 5.4 For all $n$, all $p \in \mathcal{A}(n)$, and all $p$-pure closed terms $e$

\[
\text{signs } n \text{ dopriv } p \text{ in check } p \text{ for } e = \text{ signs } n \text{ e}
\]

Proof: Let $h$ be the empty environment, for $e$ which is closed. We observe for any $n', P$:

\[
[D \vdash \text{signs } n \text{ dopriv } p \text{ in check } p \text{ for } e] n' Ph
= [D \vdash \text{dopriv } p \text{ in check } p \text{ for } e] n (\mathcal{A}(n) \cap P) h
= [D \vdash \text{check } p \text{ for } e] n ((\mathcal{A}(n) \cap P) \sqcup_n \{p\}) h
= [D \vdash \text{check } p \text{ for } e] n ((\mathcal{A}(n) \cap P) \setminus \{p\}) h
= [D \vdash e] n (\mathcal{A}(n) \cap P) h
= [D \vdash \text{signs } n \text{ e}] n' Ph
\]

In the penultimate step, two uses are needed for the lemma: to remove $p$ and, in the case that $p \in P$, to add it back.

In order to deal with the password example we need the following conditional equivalence.

Theorem 5.5 For all $n$, all $p \in \mathcal{A}(n)$, and all terms $e$

\[
\text{signs } n \text{ check } p \text{ for } e = \text{ check } p \text{ for } \text{signs } n \text{ e}
\]
Proof: We observe for any \( n', P, h \):

\[
\begin{align*}
\text{signs } n \text{ check } p \text{ for } e \text{ }] n' Ph \\
= \text{check } p \text{ for } e \text{ } n(P \cap \mathcal{A}(n)) h \\
\text{semantics}
\end{align*}
\]

\[
\begin{align*}
\text{if } p \in (P \cap \mathcal{A}(n)) \text{ then } [e] n(P \cap \mathcal{A}(n)) h \text{ else } * \\
\text{semantics}
\end{align*}
\]

The above proofs are examples of the benefit of a compositional semantics. The proofs are by direct calculation, without need for induction. For Theorem 5.4, the proof goes through for open terms as well, if the environment \( h \) is pure. One expects built-in constants to have pure and safe values.

5.2 The password example

We now revisit the password example, using Theorems 5.4 and 5.5 to eliminate checks. We abbreviate \( \text{user}, \text{root} \) as \( u, r \).

\[
passwd(\text{“mypass”}) \\
= \{ \text{because passwd} = (\text{fun } x. \text{ signs } r \text{ check } p \text{ for dopriv } w \text{ in writepass}(x))\} \\
signs r \text{ check } p \text{ for dopriv } w \text{ in writepass(“mypass”)}
\]

\[
\begin{align*}
\text{if } p \in P \text{ then } [e] n(P \cap \mathcal{A}(n)) h \text{ else } * \\
\text{sets, since } p \in \mathcal{A}(n)
\end{align*}
\]

\[
\begin{align*}
\text{check } p \text{ for signs } n \text{ e } n' Ph \text{ else } * \\
\text{semantics}
\end{align*}
\]

We now give an example of a transformation which employs a weaker notion of equality than the ones in Section 5.1, cf. (1). This is the TestGrant equation of Fournet and Gordon which, in our notation amounts to proving, for all \( n, P, h, p, e_1, e_2 \), with \( P \subseteq \mathcal{A}(n) \),

\[
[\text{test } p \text{ then } e_1 \text{ else } e_2 ] n Ph = [\text{test } p \text{ then dopriv } p \text{ in } e_1 \text{ else } e_2 ] n Ph \tag{3}
\]

5Their is slightly more general, as their test and dopriv apply to permission sets; this can be desugared to singleton permissions as in our notation.
To show (5) we use the denotational semantics and prove

\[
\text{if } p \in P \text{ then } \sem{e_1} \, \text{dopriv } p \, \sem{e_2} \quad = \quad \text{if } p \in P \text{ then } \sem{\text{dopriv } p \, e_1} \, \text{dopriv } p \, \sem{e_2}
\]

It suffices to prove that when \( p \in P \), we have \( \sem{e_1} \, \text{dopriv } p \, \sem{e_2} = \sem{\text{dopriv } p \, e_1} \, \text{dopriv } p \, \sem{e_2} \). We calculate:

\[
\begin{align*}
\text{if } p \in P \text{ then } \sem{e_1} \, \text{dopriv } p \, \sem{e_2} &= \sem{\text{dopriv } p \, e_1} \, \text{dopriv } p \, \sem{e_2} \\
&= \sem{e_1} \, P \cup_{\{p\}} \sem{e_2} \quad \text{semantics} \\
&= \sem{e_1} \, \sem{e_2} \quad \text{because } p \in P \text{ and } P \subseteq A(n) \implies p \in A(n)
\end{align*}
\]

We have proved the correctness of several other primitive equations in Fournet and Gordon’s paper [11, Section 4.1]. Specifically, Frame Frame, Frame Frame Frame, Frame Frame Grant, Frame Grant, Frame Grant Frame, Frame Grant Test, Frame Test Then, Grant Grant, Grant Frame, Grant Frame Grant and Test \( \cup \). We have also proved the correctness of their derived equations Frame Appl, Frame Frame Intersect and Frame Grant Intersect. Our proofs of these equations do not require the restriction \( P \subseteq A(n) \).

We have also proved the tail call elimination laws in [11, Section 5.2]. The basic idea in tail call elimination is to not build a new frame for the last call of a function; instead the callee can directly return to the caller’s caller. Tail call elimination is problematic with stack inspection, as a stack frame holds the principal for the current code (or, equivalently, the principal’s static permissions). As noted earlier, Clements and Felleisen [10] give an abstract machine for which tail call elimination is sound and efficient. The calculus and small-step semantics of Fournet and Gordon [11, Section 5.2] allows a limited modeling of tail call elimination. Here is one of their two transformations, in our notation:

\[
signs(n_2)((\text{fun } x. \, \text{signs } n_1 \, e_1) \, e_2) = ((\text{fun } x. \, \text{signs } n_1 \, e_1) \, e_2)
\]

(4)

From left to right this can be read as dropping the “frame” of the calling context. In their setting it can be read as a transition, provided that \( e_2 \) is a value. They show that this added transition is admissible, in the sense of not changing outcomes, provided that the callee’s static permissions are among the caller’s, \( i.e. A(n_1) \subseteq A(n_2) \). For our purposes, a value is a boolean literal, a variable (whose value is thus in the environment) or an abstraction. We shall prove that the equation holds in our semantics, under these conditions. For the proof, it is convenient to use the following easily proved fact which holds for any \( D, e_1, e_2, n, P, h \):

\[
\sem{D \vdash (\text{fun } x. \, e_1) \, e_2} = \sem{D, x : t \vdash e_1} \, P \sem{h \mid x \mapsto \sem{D \vdash e_2}}
\]

(5)

Note: if \( e_2 \) is a value then its semantics \( \sem{D \vdash e_2} \) is independent from \( P \) (though not necessarily from \( n \) or \( h \)); see Fig. 2. To prove (4) we observe

\[
\begin{align*}
\sem{D \vdash \text{signs } n_2 ((\text{fun } x. \, \text{signs } n_1 \, e_1) \, e_2)} &= \sem{D \vdash ((\text{fun } x. \, \text{signs } n_1 \, e_1) \, e_2)} \, n(P \cap A(n_2)) \quad \text{by semantics of signs} \\
&= \sem{D, x : t \vdash \text{signs } n_1 \, e_1} \, n(P \cap A(n_2)) \sem{h \mid x \mapsto \sem{D \vdash e_2} \, n(P \cap A(n_2)) \sem{h}} \quad \text{lemma (5)} \\
&= \sem{D, x : t \vdash e_1} \, n(P \cap A(n)) \sem{h \mid x \mapsto \sem{D \vdash e_2} \, n(P \cap A(n)) \sem{h}} \quad \text{sem., } A(n_1) \subseteq A(n_2) \\
&= \sem{D, x : t \vdash e_1} \, n(P \cap A(n)) \sem{h \mid x \mapsto \sem{D \vdash e_2} \, n(P \cap A(n)) \sem{h}} \quad \text{e_2 value, Note above} \\
&= \sem{D \vdash \text{signs } n_2 ((\text{fun } x. \, \text{signs } n_1 \, e_1) \, e_2)} \quad \text{semantics of signs} \\
&= \sem{D \vdash ((\text{fun } x. \, \text{signs } n_1 \, e_1) \, e_2)} \quad \text{lemma (5)}
\end{align*}
\]

As with most of the other equations, the restriction \( P \subseteq A(n) \) is not necessary here. Fournet and Gordon give a second equation, also provable in our setting, that models tail calls involving dopriv.

---

For a few we need to consider the evident generalization of our language that allows permission sets in dopriv (for Grant Grant, Grant Frame, and Frame Grant Intersect) and in test (for Test \( \cup \)).
6 Using the Static Analysis

Section 5 gives several program transformations that can be justified by the eager denotational semantics of our language. A more drastic transformation is possible under some conditions. The safety results of Section 4 show that if the static analysis derives a judgement \( \Delta \vdash e : \theta, \Pi \), then executing \( e \) using a privilege set that contains at least the enabled privileges \( \Pi \) would not lead to a security error. We should therefore be able to drop all \( \text{dopriv}'s \) and \( \text{check}'s \) from \( e \). If \( e \) is test-free, we can then show that the meaning of \( e \) is the same as its meaning with \( \text{dopriv}'s \) and \( \text{check}'s \) erased. This is formalized below.

**Definition 6.1** The erasure translation \((\cdot)^-\) is defined as follows:

\[
\begin{align*}
\text{true}^- & = \text{true} \\
\text{x}^- & = x \\
(\text{if } e_1 \text{ then } e_2 \text{ else } e_3)^- & = \text{if } e_1^- \text{ then } e_2^- \text{ else } e_3^- \\
(\text{fun } x. e^-) & = \text{fun } x. e^- \\
(\text{letrec } f(x) = e_1 \in e_2)^- & = \text{letrec } f(x) = e_1^- \in e_2^- \\
(\text{signs } n e)^- & = \text{signs } n e^- \\
(\text{dopriv } p \in e)^- & = e^- \\
(\text{check } p \text{ for } e)^- & = e^- \\
(\text{test } p \text{ then } e_1 \text{ else } e_2)^- & \text{ is undefined.}
\end{align*}
\]

**Theorem 6.2** Let \( e \) be test-free and let \( \varnothing; n \vdash e : \text{bool}, \Pi \). Then for all \( P \in \mathcal{P}(\text{Privileges}) \), if \( \Pi \subseteq P \) then \( \llbracket \varnothing \vdash e : \text{bool} \rrbracket nP \{ \} = \llbracket \varnothing \vdash e^- : \text{bool} \rrbracket nP \{ \} \).

**Proof:** Immediate consequence of Lemma 6.6 and definition rel bool below. ■

**Definition 6.3** For each annotated type \( \theta \) the relation \( \text{rel } \theta \) on \( \llbracket \theta^* \rrbracket_{\downarrow, \ast} \) is defined as follows: For all \( \theta \), \( \text{rel } \theta \downarrow \downarrow \) holds and otherwise \( \text{rel } \theta \downarrow d \downarrow d' \) is false if either \( d \) or \( d' \) is in \( \{ \downarrow, \ast \} \). For values other than \( \downarrow, \ast \), the definition is by induction on structure of \( \theta \).

\[
\begin{align*}
\text{rel } \theta \downarrow \text{bool } b \downarrow b' & \iff b = b' \\
\text{rel } (\theta_1 \rightarrow \theta_2) \downarrow f \downarrow f' & \iff \forall P \in \mathcal{P}(\text{Privileges}), \forall d, d' \in \llbracket \theta_1^* \rrbracket, \\
& \Pi \subseteq P \land \text{rel } \theta_1 \downarrow d \downarrow d' \Rightarrow \text{rel } \theta_2 (fPd) (f' Pd')
\end{align*}
\]

For annotated type environment \( \Delta \), the predicate \( \text{rel } \Delta \) on \( \llbracket \Delta^* \rrbracket \) is defined by \( \text{rel } \Delta h \downarrow h' \iff \text{dom}(h) = \text{dom}(h') \) and \( \forall x \in \text{dom}(h). \text{rel } (\Delta x) (h.x) (h'.x) \).

**Fact 6.4** \( \theta \leq \theta' \) and \( \text{rel } \theta \downarrow d \downarrow d' \) imply \( \text{rel } \theta' \downarrow d \downarrow d' \).

**Proof:** By induction on derivation of \( \theta \leq \theta' \). The result is clear for \( \text{bool } \leq \text{bool} \). For \( \theta_1 \rightarrow \theta_2 \leq (\theta_1' \rightarrow \theta_2') \), assume \( \text{rel } (\theta_1 \rightarrow \theta_2) f \downarrow f' \). To show \( \text{rel } (\theta_1' \rightarrow \theta_2') f \downarrow f' \), consider any \( P \in \mathcal{P}(\text{Privileges}) \), such that \( \Pi' \subseteq P \), and any \( d, d' \in \llbracket \theta_1^* \rrbracket \) with \( \text{rel } \theta_1' \downarrow d \downarrow d' \). From the subtyping, we know that \( \Pi \subseteq \Pi' \), hence \( \Pi \subseteq P \). Moreover, by induction on derivation of \( \theta_1' \leq \theta_1 \), we obtain \( \text{rel } \theta_1' \downarrow d \downarrow d' \) implies \( \text{rel } \theta_1 \downarrow d \downarrow d' \). Hence from assumption \( \text{rel } (\theta_1 \rightarrow \theta_2) f \downarrow f' \), we obtain \( \text{rel } \theta_2 (fPd) (f' Pd') \). Now by induction on derivation \( \theta_2 \leq \theta_2' \), we obtain \( \text{rel } \theta_2' (fPd) (f' Pd') \). ■
**Fact 6.5** The relation rel preserves lubs. That is, for any \( \theta \), let \( u, u' : \mathbb{N} \rightarrow [\theta^*] \) be ascending chains. Then, \( \forall i. \text{rel } \theta \ u_i \ u'_i \) implies rel \( \theta \ (\sqcup_i u_i) (\sqcup_i u'_i) \).

**Proof:** By structural induction on \( \theta \). When \( \theta = \text{bool} \), we have \( \sqcup_i u_i = \sqcup_i u'_i \) = true or false or \( \bot \). Thus the result holds by definition rel.

When \( \theta = (\theta_1 \rightarrow \theta_2) \), assume \( P \in \mathcal{P} \) (Privileges) and \( d, d' \in [\theta^*] \), such that \( \Pi \subseteq P \) and rel \( \theta_1 \ d \ d' \). Then, from assumption rel \( (\theta_1 \rightarrow \theta_2) \ u_i \ u'_i \) we obtain rel \( \theta_2 \ (u_i Pd) (u'_i Pd') \) for every \( i \). Hence, by the induction hypothesis on \( \theta_2 \), we get rel \( \theta_2 \ (\sqcup_i(u_i Pd)) (\sqcup_i(u'_i Pd')) \). Because lubs are pointwise, we get rel \( \theta_2 \ (\sqcup_i u_i) Pd) (\sqcup_i u'_i) Pd') \).

**Lemma 6.6** Suppose \( \Delta; n \vdash e : \theta, \Pi \) is derivable and \( e \) is test-free. Then for all \( P \in \mathcal{P} \) (Privileges), for all \( h, h^- \in [\Delta]^\star \), if rel \( \Delta \ h \ h^- \) and \( \Pi \subseteq P \) then rel \( \theta \ u \ u^- \), where \( u = [\Delta^\star \vdash e : \theta^*]nPh \) and \( u^- = [\Delta^\star \vdash e^- : \theta^*]nPh^- \).

(Note that \( h^- \), \( u^- \) are just suggestively named identifiers whereas \( e^- \) is the erasure of \( e \).) Theorem 6.2 follows from the lemma because rel \( \emptyset \ { } \emptyset \) and by definition rel \( \text{bool} \), \( [\emptyset \vdash e : \text{bool}]nP \{ \} = [\emptyset \vdash e^- : \text{bool}]nP \{ \} \).

**Proof:** of Lemma. Go by induction on the typing derivation, \( \Delta; n \vdash e : \theta, \Pi \). Throughout, we assume \( P \in \mathcal{P} \) (Privileges) and \( h, h^- \in [\Delta]^\star \) and rel \( \Delta \ h \ h^- \). Let \( u = [\Delta^\star \vdash e : \theta^*]nPh \) and \( u^- = [\Delta^\star \vdash e^- : \theta^*]nPh^- \) for each case of \( e \).

- Case true: Then, \( u = \text{true} \ = u^- \) and rel \( \text{bool} \ u \ u^- \) by definition rel.
- Case x: Then, \( u = h.x \) and \( u^- = h^- . x \). And, rel \( \theta \ u \ u^- \) follows from assumption rel \( \Delta \ h \ h^- \).
- Case if \( e \) then \( e_1 \) else \( e_2 \): Then \( \Pi_1 \cup \Pi_2 \cup \Pi_3 \subseteq P \), and

\[
\begin{align*}
  u & = \text{if } b \text{ then } [\Delta^\star \vdash e_1 : \theta^*]nPh \text{ else } [\Delta^\star \vdash e_2 : \theta^*]nPh \\
  u^- & = \text{if } b^- \text{ then } [\Delta^\star \vdash e^- : \theta^*]nPh^- \text{ else } [\Delta^\star \vdash e^- : \theta^*]nPh^- 
\end{align*}
\]

where \( b = [\Delta^\star \vdash e : \text{bool}]nPh \) and \( b^- = [\Delta^\star \vdash e^- : \text{bool}]nPh^- \). By the induction hypothesis on the typing derivation of \( e \), noting that \( \Pi_1 \subseteq P \), we have rel \( \text{bool} \ b \ b^- \). If \( b = \bot = b^- \) then \( u = \bot = u^- \) and rel \( \theta \bot \bot \). Otherwise, \( b = \text{true} \) or \( b = \text{false} \). In the former case, by the induction hypothesis on the typing derivation of \( e_1 \), noting that \( \Pi_2 \subseteq P \), we have rel \( \theta \ u \ u^- \). In the latter case, by the induction hypothesis on the typing derivation of \( e_2 \), noting that \( \Pi_3 \subseteq P \), we have rel \( \theta \ u \ u^- \).

- Case \( \text{fun } x . e \): Then \( u = \lambda P'. \lambda d. \ [\Delta^\star \vdash x : \theta_1^* \vdash e : \theta_2^*]nP' [h | x \mapsto d] \)\( u^- = \lambda P'. \lambda d^- . \ [\Delta^\star \vdash x : \theta_1^* \vdash e^- : \theta_2^*]nP'[h^- | x \mapsto d^-] \)

To prove rel \( (\theta_1 \rightarrow \theta_2) \ u \ u^- \), consider any \( P' \in \mathcal{P} \) (Privileges) and any \( d, d^- \in [\theta^*] \) such that \( \Pi \subseteq P' \) and rel \( \theta_1 \ d \ d^- \), to show rel \( \theta_2 \ (uP'd') (u^-P'd^-) \). By semantics,

\[
\begin{align*}
  uP'd & = [\Delta^\star \vdash x : \theta_1^* \vdash e : \theta_2^*]nP' [h | x \mapsto d] \\
  u^-P'd^- & = [\Delta^\star \vdash x : \theta_1^* \vdash e^- : \theta_2^*]nP'[h^- | x \mapsto d^-] 
\end{align*}
\]

So the induction hypothesis for \( e \) yields rel \( \theta_2 \ (uP'd') (u^-P'd^-) \) provided that \( \Pi \subseteq P' \) and rel \( (\Delta, x : \theta_1)[h | x \mapsto d][h^- | x \mapsto d^-] \). We have \( \Pi \subseteq P' \) by assumption, and rel \( (\Delta, x : \theta_1)[h | x \mapsto d][h^- | x \mapsto d^-] \) follows from rel \( \Delta \ h \ h^- \) and rel \( \theta_1 \ d \ d^- \).
• Case $e_1$: Let $f = \llbracket \Delta \vdash e_1 : (\theta_1 \rightarrow \theta_2)^* \rrbracket nPh$ and $d = \llbracket \Delta \vdash e_2 : \theta_3^* \rrbracket nPh$, so that $u = fPd$. Let $f' = \llbracket \Delta \vdash e_2 : (\theta_1 \rightarrow \theta_2)^* \rrbracket nPh$ and $d' = \llbracket \Delta \vdash e_2 : \theta_3^* \rrbracket nPh$, so that $u' = f'Pd'$. (Recall that $\theta'_1 \leq \theta_1$ implies $\theta'^*_1 = \theta_1^*$ so the applications $fPd$ and $f'Pd'$ make sense.) From rel $\Delta h h'$ and assumption $\Pi \cup \Pi_1 \cup \Pi_2 \subseteq P$, we get by induction on $e_1$ that rel $(\theta_1 \rightarrow \theta_2) f f'$, and we get rel $\theta'_1 d d'$ by induction on $e_2$. By $\theta'_1 \leq \theta_1$ and Fact 6.4, we have rel $\theta'_1 d d'$, and by definition rel $(\theta_1 \rightarrow \theta_2) f f'$, since $\Pi \subseteq P$, we get rel $\theta_2 (fPd)(f'Pd')$.

• Case letrec $f(x) = e_1$ in $e_2$: Then, $\Pi \cup \Pi_1 \subseteq P$.

Now $u = \llbracket \Delta, f : (\theta_1 \rightarrow \theta_2)^* \rrbracket nP[h \mapsto \text{fix } G]$ $u' = \llbracket \Delta, f : (\theta_1 \rightarrow \theta_2)^* \rrbracket nP[h' \mapsto \text{fix } G']$ where $G(g) = \lambda P'. \lambda d. \llbracket \Delta, f : (\theta_1 \rightarrow \theta_2)^*, x : \theta_1^* \vdash e_1 : \theta_2^* \rrbracket nP[h \mapsto g, x \mapsto d]$ $G'(g') = \lambda P'. \lambda d'. \llbracket \Delta, f : (\theta_1 \rightarrow \theta_2)^*, x : \theta_1^* \vdash e_1 : \theta_2^* \rrbracket nP'[h' \mapsto g', x \mapsto d']$ To show rel $\theta u u'$ by induction on $e_2$, we need $\Pi \subseteq P$ and

$$\text{rel } \Delta, f : \theta_1 \rightarrow \theta_2 [h \mapsto \text{fix } G] [h' \mapsto \text{fix } G']$$

The former follows from assumption $\Pi \cup \Pi_1 \subseteq P$. The latter follows from assumption, rel $\Delta h h'$, and rel $(\theta_1 \rightarrow \theta_2) (fPd')(f'Pd')$, which we now proceed to show.

Now fix $G = \bigsqcup_i g_i$, where $g_0 = \lambda P'. \lambda d \in [\theta_1^*], \bot$ and $g_{i+1} = G(g_i)$. Also fix $G' = \bigsqcup_i g'_i$, where $g_0' = \lambda P'. \lambda d' \in [\theta_1^*], \bot$ and $g_{i+1}' = G'(g'_i)$. And, rel $(\theta_1 \rightarrow \theta_2) (fPd')(f'Pd')$ is a consequence of the following claim:

$$\forall i. \text{rel } (\theta_1 \rightarrow \theta_2) g_i g_i'$$

Then from Lemma 6.5 we get rel $(\theta_1 \rightarrow \theta_2) (\bigsqcup_i g_i)(\bigsqcup_i g_i')$. It remains to show (6), for which we proceed by induction on $i$.

Base case: Show rel $(\theta_1 \rightarrow \theta_2) g_0 g_0'$. Assume any $P' \in \mathcal{P}(\text{Privileges})$ and any $u, v \in [\theta_1^*]$, such that $\Pi \subseteq P'$ and rel $\theta_1 u v$. Then $g_0 P' u = \bot = g_0 P' v$ and rel $\theta_2 (g_0 P' v)(g_0 P' u)$.

Induction step: Assume rel $(\theta_1 \rightarrow \theta_2) g_i g_i'$, to show rel $(\theta_1 \rightarrow \theta_2) g_{i+1} g_{i+1}'$.

Now $g_{i+1} = \lambda P'. \lambda d. \llbracket \Delta, f : (\theta_1 \rightarrow \theta_2)^*, x : \theta_1^* \vdash e_1 : \theta_2^* \rrbracket nP[h \mapsto g_i, x \mapsto d]$ $g_{i+1}' = \lambda P'. \lambda d'. \llbracket \Delta, f : (\theta_1 \rightarrow \theta_2)^*, x : \theta_1^* \vdash e_1 : \theta_2^* \rrbracket nP'[h' \mapsto g_i', x \mapsto d']$

Assume any $P' \in \mathcal{P}(\text{Privileges})$ and $u, v \in [\theta_1^*]$, such that $\Pi \subseteq P'$ and rel $\theta_1 u v$. Then

$g_{i+1} P' u = \llbracket \Delta, f : (\theta_1 \rightarrow \theta_2)^*, x : \theta_1^* \vdash e_1 : \theta_2^* \rrbracket nP'[h \mapsto g_i, x \mapsto v]$ $g_{i+1}' P' v = \llbracket \Delta, f : (\theta_1 \rightarrow \theta_2)^*, x : \theta_1^* \vdash e_1 : \theta_2^* \rrbracket nP'[h' \mapsto g_i', x \mapsto v]$ Note that rel $(\Delta, f : \theta_1 \rightarrow \theta_2, x : \theta_1) [h \mapsto g_i, x \mapsto v] [h' \mapsto g_i', x \mapsto v]$. Therefore, by the main induction hypothesis on the typing derivation $\Delta, f : \theta_1 \rightarrow \theta_2, x : \theta_1; n \vdash e_1 : \theta_2$, since $\Pi \subseteq P$, we obtain rel $\theta_2 (g_{i+1} P' v)(g_{i+1}' P' u)$.

• Case $\text{signs } n' e$: Then $\Pi \subseteq P$ and $u = \llbracket \Delta \vdash e : \theta^* \rrbracket n'(P \cap \mathcal{A}(n'))h$. The induction hypothesis on the typing derivation of $e$ can be used to obtain rel $\theta u u'$, because $\Pi \subseteq (P \cap \mathcal{A}(n'))$ which follows from assumption $\Pi \subseteq P$ and side condition $\Pi \subseteq \mathcal{A}(n')$.

• Case dopriv $p$ in $e$: Then $\Pi \subseteq P$ and $u = \llbracket \Delta \vdash e : \theta^* \rrbracket n(P \cup \{p\})h$. By the induction hypothesis for $e$, noting that $(\Pi \cup \{p\}) \subseteq (P \cup \{p\})$, we have rel $\theta u \llbracket \Delta \vdash e : \theta^* \rrbracket n(P \cup \{p\})h$. But now $e$ is $p$-pure. So by Lemma 5.3 $\llbracket \Delta \vdash e : \theta^* \rrbracket n(P \cup \{p\})h = \llbracket \Delta \vdash e : \theta^* \rrbracket nPh$. But $u' = \llbracket \Delta \vdash e : \theta^* \rrbracket nPh$. Hence rel $\theta u u'$.
Case check $p$ for $e$:

Then $\Pi \cup \{p\} \subseteq P$, hence $p \in P$. Now $u = \text{if } p \in P \text{ then } [[\Delta^* \vdash e : \theta^*]]vP \text{ else } \ast$. Since $p \in P$, we have, $u = [[\Delta^* \vdash e : \theta^*]]vP$ and, by the induction hypothesis on the typing derivation of $e$, we have $\text{rel } \theta \ u \ [[\Delta^* \vdash e^\ast : \theta^\ast]]vP$. Hence $\text{rel } \theta \ u^\ast$.

7 Stack Semantics

This section gives a formal semantics using stack inspection, and shows that for standard expressions it coincides with the eager semantics. The connection is much more direct than that of Wallach, Appel and Felten, so a complete detailed proof is not very lengthy.

Because the operations on the stack are in fact stack-like, it is straightforward to give a denotational style semantics parameterized on the stack. We define $\text{Stacks} = \text{nonempty list of (Principals } \times \ P\text{Privileges), taken as a cpo ordered by equality. The top is the head of the list, and we write infix :: for cons, so } (n, P) :: S \text{ is the stack with } (n, P) \text{ on top of } S, \text{ as in Section 2. We also use the predicate } \text{chk defined there, and recall the definition } p \in \text{privs } S \iff \text{ chk}(p,S)$.

Fact 7.1 For all $S$ and all $n$ we have $\text{privs}(S) \cap \mathcal{A}(n) = \text{privs}((n, \emptyset) :: S)$.

Proof: The sets are equal because for any $p$

$$p \in \text{privs}((n, \emptyset) :: S) \iff \text{chk}(p, (n, \emptyset) :: S) \iff p \in \mathcal{A}(n) \wedge \text{chk}(p, S) \iff p \in \mathcal{A}(n) \wedge p \in \text{privs}(S) \text{ by def privs.}$$

The stack semantics of an expression is a function

$$([[D \vdash e : t]]) \in \text{Stacks} \to ([[D]]) \to ([[t]])_{\perp^s}$$

Just as in the eager semantics, we need to account for dynamic binding of privileges by interpreting arrow types using an extra parameter. The stack semantics of types is as follows.

$$([[\text{bool}]]) = \{\text{true, false}\}$$

$$([[t_1 \to t_2]]) = \text{Stacks} \to ([[t_1]]) \to ([[t_2]])_{\perp^s}$$

The semantics of expressions is in Figure 4.

We can now relate the denotational semantics of Figure 2 to the stack semantics of Figure 3.

Theorem 7.2 (Consistency) For any standard expression $e$ and stack $(n, P) :: S$, we have

$$[[\emptyset \vdash e : \text{bool}]]vP = ([[\emptyset \vdash e : \text{bool}))((n, P') :: S)\} \text{ where } P = \text{privs}((n, P') :: S).$$

Proof: Immediate consequence of Lemma 7.5 and definition sim bool below.

As in the proof of safety, we need to generalize the result to allow nonempty contexts. We also consider expressions of arrow type, for which a logical relation is needed.

Definition 7.3 Define data-type indexed family $\text{sim } t \subseteq [[t]]_{\perp^s} \times [[t]]_{\perp^s}$ as follows. For any $t$, $\text{sim } t \ d \ d'$ is true if $d = d'$ and $d \in \{\perp, \ast\}$; it is false if $d \neq d'$ and $d$ or $d'$ is in $\{\perp, \ast\}$. Otherwise:

$$\text{sim } \text{bool } b \ b' \iff b = b'$$

$$\forall S \in \text{Stacks.} \forall d \in [[t_1]], \forall d' \in [[t_1]]. \text{sim } t_1 \ d \ d' \Rightarrow \text{sim } t_2 (f \ (\text{privs } S) \ d) (f' \ S \ d')$$
The Consistency Theorem follows from the lemma because

Proof: A. Banerjee & D. A. Naumann

(305)

\[
\text{The relation } \bullet \text{ Cases }
\]

When:

\[
\text{if } b \text{ then } (D' \vdash e : t_1)Sh \text{ else } (D' \vdash e_2 : t_2)Sh
\]

\[
(\lambda S' \in \text{Stacks. } \lambda d \in (t_1)).
\]

\[
(\lambda x : t_1 \vdash e_2 : t_2)Sh'[h \mid x \mapsto d]
\]

\[
(\lambda x : t_1 \vdash e_2 : t_2)Sh'[h' \mid x \mapsto d']
\]

To show sim \((t_1 \rightarrow t_2) u u'\), need to show that for any \(S'', d'', d'''\), such that sim \(t_1 d'' d'''\), it is the

\[
(\lambda \text{check } p \text{ for } e : t)Sh = \text{if chk}(p, S) \text{ then } (D' \vdash e : t)Sh \text{ else }^*
\]

\[
(\lambda \text{letrec } f(x) = e_1 \text{ in } e_2 : t_2)Sh
\]

\[
= \text{let } G(g) = \lambda S'. \lambda d. (D, f : t_1 \rightarrow t_2, x : t_1 \vdash e : t_2)S[h \mid f \mapsto g, x \mapsto d] \text{ in}
\]

\[
(\lambda x : t_1 \rightarrow t_2 \vdash e_2 : t_2)Sh' = \text{let } G = (D, f : t_1 \rightarrow t_2, x : t_1 \vdash e : t_2)S[h \mid f \mapsto \text{fix } G]
\]

\[
(\lambda \text{dopriv } p \in e : t)Sh = \text{if } \text{chk}(p, S) \text{ then } (D' \vdash e : t)Sh \text{ else }^*
\]

\[
(\lambda \text{test } p \text{ then } e_1 \text{ else } e_2 : t_2)Sh
\]

\[
= \text{let } G = (D, f : t_1 \rightarrow t_2, x : t_1 \vdash e : t_2)S[h \mid f \mapsto \text{fix } G]
\]

\[
(\lambda x : t_1 \rightarrow t_2 \vdash e_2 : t_2)Sh' = \text{let } G = (D, f : t_1 \rightarrow t_2, x : t_1 \vdash e : t_2)S[h \mid f \mapsto \text{fix } G]
\]

Figure 4: Stack semantics

An environment \(h \in [D]\) simulates an environment \(h' \in [D]\), written sim \(D h h'\), provided sim \((D, x) (h.x) (h'.x)\) for all \(x \in \text{dom}(h)\).

Lemma 7.4 The relation sim preserves lbs. That is, for any \(t\), if \(u : \mathbb{N} \rightarrow [t]\) and \(u' : \mathbb{N} \rightarrow [t]\) are ascending chains and \(\forall i. \text{sim } t u_i u'_i\) then sim \(t \uparrow_i u_i (t \uparrow_i u'_i)\).

Proof: By structural induction on \(t\). Assume that sim \(t u_i u'_i\). When \(t = \text{bool}\), by definition sim we obtain, for each \(i\), sim \(u_i = u'_i\). Thus sim \(t (\uparrow_i u_i) (\uparrow_i u'_i)\).

When \(t = t_1 \rightarrow t_2\), consider any \(P, S, d, d'\) with \(P = \text{privs}(S)\) and sim \(t_1 d d'\). We must show sim \(t_2 (\uparrow_i u_i P d) (\uparrow_i u'_i S'd')\), i.e., by definition of lbs we must show, sim \(t_2 [u_i P d] (u'_i S'd')\). By assumption, for every \(i\), sim \((t_1 \rightarrow t_2) u_i u'_i\), hence, sim \(t_2 (u_i P d) (u'_i S'd')\) holds for each \(i\). Therefore, by induction for \(t_2\), we obtain sim \(t_2 (\uparrow_i (u_i P d) (\uparrow_i (u'_i S'd'))\).

Lemma 7.5 For any stack \((n, P') :: S)\), for any standard expression \(e\), and any \(D, t, h, h'\), let \(u = [D \vdash e : t]nPh\) where \(P = \text{privs}((n, P') :: S)\), and let \(u' = [(D \vdash e : t)](n, P') :: S)h'\). Then sim \(D h h' \Rightarrow h = u = u'\).

The Consistency Theorem follows from the lemma because sim \(\text{\{} \text{\{}\} \text{\}}\) and since sim \text{bool} \(u u'\) implies \(u = u'\).

Proof: of Lemma. Go by induction on \(e\).

- Cases \text{true} and \(x\): Immediate from semantic definitions.
- Case if \(e\) then \(e_1\) else \(e_2\): Directly by induction.
- Case \text{fun } x. e: Let \(u = [D \vdash \text{fun } x. e : t_1 \rightarrow t_2]nPh\) and let
  
  \[
  u' = ([D \vdash \text{fun } x. e : t_1 \rightarrow t_2])Sh'
  \]

Then \(u = \lambda P'. \lambda d. [D, x : t_1 \vdash e : t_2]nP'[h \mid x \mapsto d]\)

\[
= \lambda S'. \lambda d. ([D, x : t_1 \vdash e : t_2]S'[h' \mid x \mapsto d'])
\]

To show sim \((t_1 \rightarrow t_2) u u'\), need to show that for any \(S'', d'', d'''\), such that sim \(t_1 d'' d'''\), it is the
case that \(\text{sim} \; t_2 \; (u \; \text{prvs} \; S'') \; (u' \; S'' \; d'')\). By standardness, \(e\) is \(\text{signs} \; n' \; e'\) for some \(n', e'\).

Thus we can proceed as follows, using \(e \equiv \text{signs} \; n' \; e'\) and semantics of \(\text{signs}\).

\[
\begin{align*}
  u \; \text{prvs} \; S'' \; d'' &= \left[\lambda d. x : t_1 \vdash e : t_2\right] n \; \text{prvs} \; S'' \left[\lambda x \rightarrow d''\right] \\
  &= \left[\lambda d. x : t_1 \vdash e' : t_2\right] n' \; \text{prvs}(S'') \cap \mathcal{A}(n') \left[\lambda x \rightarrow d''\right] \\
  u' \; S'' \; d'' &= \left[\lambda d. x : t_1 \vdash e : t_2\right] S'' \left[\lambda x \rightarrow d''\right] \\
  &= \left[\lambda d. x : t_1 \vdash e' : t_2\right] \left(\langle n', \emptyset \rangle \; :: \; S''\right) \left[\lambda x \rightarrow d''\right]
\end{align*}
\]

Note that by definition \(\text{sim}\) and by assumption \(\text{sim} \; t_1 \; d'' \; d''\), we have, \(\text{sim} \; (D, x : t_1) \left[\lambda x \rightarrow d''\right] \left[\lambda x \rightarrow d''\right].\) Furthermore, by Fact \(7.1\) \(\text{prvs}(S'') \cap \mathcal{A}(n') = \text{prvs}(\langle n', \emptyset \rangle :: S'')\). Therefore, by induction for \(e'\), we obtain, \(\text{sim} \; t_2 \; (u \; \text{prvs} \; S'') \; (u' \; S'' \; d'')\). This is where we need Definition \(2.1\).

- **Case** \(e_1 \; e_2\): \(\left[D \vdash e_1 \; e_2 : t_2\right] n \text{Ph} = \) \[
\begin{align*}
  \left[D \vdash e_1 \; e_2 : t_2\right] n \text{Ph} &= \text{let } f = \left[D \vdash e_1 : t_1 \rightarrow t_2\right] n \text{Ph} \quad \text{in} \\
  &\quad \left[D \vdash e_2 : t_1\right] n \text{Ph} \quad \text{in} \quad \text{fpd} \\
  \left[D \vdash e_1 \; e_2 : t_2\right] \text{Sh} &= \text{let } f' = \left[D \vdash e_1 : t_1 \rightarrow t_2\right] \text{Sh} \quad \text{in} \\
  &\quad \left[D \vdash e_2 : t_1\right] \text{Sh} \quad \text{in} \quad \text{fpd}\text{d'}
\end{align*}
\]

Need to show \(\text{sim} \; t_2 \; (\text{fpd}) \; (f' \; \text{sd'})\). Since \(\text{sim} \; D \; h'\) and \(P = \text{prvs}(S)\), therefore, by induction for \(e_1\), we have \(\text{sim} \; (t_1 \rightarrow t_2) \; f \; f'\). Similarly, by induction for \(e_2\), we have \(\text{sim} \; t_1 \; d \; d'\). Hence the result follows by definition \(\text{sim}\) since \(P = \text{prvs}(S)\). This case of the proof shows the necessity of defining the relation \(\text{sim}\).

- **Case** \(\text{letrec } f(x) = e_1 \; \text{in} \; e_2\):

\[
\begin{align*}
  \left[D \vdash \text{letrec } f(x) = e_1 \; \text{in} \; e_2 : t\right] n \text{Ph} &= \text{let } g = \lambda P'. \lambda d. \left[D, f : t_1 \rightarrow t_2, x : t_1 \vdash e_1 : t_2\right] n \text{Ph} \; [h \; | \; f \rightarrow g, x \rightarrow d] \quad \text{in} \\
  &\quad \left[D, f : t_1 \rightarrow t_2 \vdash e_2 : t\right] n \text{Ph} \; [h \; | \; f \rightarrow \text{fix } G] \\
  \left[D \vdash \text{letrec } f(x) = e_1 \; \text{in} \; e_2 : t\right] \text{Sh} &= \text{let } g' = \lambda S'. \lambda d'. \left[D, f : t_1 \rightarrow t_2, x : t_1 \vdash e_1 : t_2\right] S' \; [h' \; | \; f \rightarrow g', x \rightarrow d'] \quad \text{in} \\
  &\quad \left[D, f : t_1 \rightarrow t_2 \vdash e_2 : t\right] S' \; [h' \; | \; f \rightarrow \text{fix } G']
\end{align*}
\]

To show the result, it suffices to show \(\text{sim} \; (t_1 \rightarrow t_2) \; (\text{fix } G) \; (\text{fix } G')\), because then we can use induction for \(e_2\), noting that \(\text{sim} \; (D, f : t_1 \rightarrow t_2) \; [h \; | \; f \rightarrow \text{fix } G] \; [h' \; | \; f \rightarrow \text{fix } G'],\) and that \(P = \text{prvs}(S)\). Accordingly, we demonstrate the following claim:

\[
\forall i. \text{sim} \; (t_1 \rightarrow t_2) \; g_i \; g'_i\quad \text{(7)}
\]

Then from Lemma \(7.4\) we get \(\text{sim} \; (t_1 \rightarrow t_2) \; [\lambda x \; g_i \; g'_i]\). This completes the proof. To show \((7)\), we proceed by induction on \(i\). We have:

\[
\begin{align*}
  g_0 &= \lambda P'. \lambda d. \perp \\
  g_{i+1} &= \lambda P'. \lambda d. \left[D, f : t_1 \rightarrow t_2, x : t_1 \vdash e_1 : t_2\right] n \text{Ph} \; [h \; | \; f \rightarrow g_i, x \rightarrow d] \\
  &= \{\text{because } e_1 \equiv \text{signs} \; n' \; e'_i \text{ by standardness}\} \\
  &= \lambda P'. \lambda d. \left[D, f : t_1 \rightarrow t_2, x : t_1 \vdash e_1 : t_2\right] n' \; \text{prvs}(P' \cap \mathcal{A}(n')) \; [h \; | \; f \rightarrow g_i, x \rightarrow d] \\
  g'_0 &= \lambda S'. \lambda d'. \perp \\
  g'_{i+1} &= \lambda S'. \lambda d'. \left[D, f : t_1 \rightarrow t_2, x : t_1 \vdash e_1 : t_2\right] S' \; [h' \; | \; f \rightarrow g'_i, x \rightarrow d'] \\
  &= \{\text{because } e_1 \equiv \text{signs} \; n' \; e'_i\} \\
  &= \lambda S'. \lambda d'. \left[D, f : t_1 \rightarrow t_2, x : t_1 \vdash e_1 : t_2\right] \left(\langle n', \emptyset \rangle \; :: \; S'\right) \; [h' \; | \; f \rightarrow g'_i, x \rightarrow d']
\end{align*}
\]
The second is logical relations, defined by structural recursion on types. We are glad for the opportunity.

The first is proof of program equalities via equational reasoning on denotations. Our work was motivated by the hope, inspired by discussions with Dave Schmidt, for more principled operational semantics. The hope, inspired by discussions with Dave Schmidt, for more principled operational semantics. The hope, inspired by discussions with Dave Schmidt, for more principled operational semantics.

8 Conclusion

Clearly, \( \text{sim} \left( t_1 \rightarrow t_2 \right) g_0 g_0' \), by definition \( \text{sim} \). To show \( \text{sim} \left( t_1 \rightarrow t_2 \right) g_{i+1} g_i' \), assume \( \text{sim} \left( t_1 \rightarrow t_2 \right) g_i g_i' \) (induction hypothesis), and that for any \( S' \) and \( P' = \text{privs}(S') \), \( \text{sim} \left( t_1 \rightarrow t_2 \right) g_i g_i' \) holds. Then

\[
\text{sim} \left( D, f : t_1 \rightarrow t_2, x : t_1 \right) [h \mid f \mapsto g_i, x \mapsto d] [h' \mid f \mapsto g_i', x \mapsto d']
\]

by definition \( \text{sim} \) and since \( \text{sim} D h h' \). Now by Fact[7.1] \( P' \cap A(n') = \text{privs}(A(n') : S') \), so by the main induction hypothesis on \( e_1 \), \( \text{sim} \left( t_2 \rightarrow t_2 \right) g_{i+1} g_{i+1} \) holds.

- Case \( \text{signs} \ n \ e \): We have: \( \left[ D \vdash \text{signs} n' e : t \right] \text{Ph} = \left[ D \vdash e : t \right] n'(P \cap A(n'))h \) and \( \left[ D \vdash \text{signs} n' e : t \right] \text{Sh} = \left( \left[ D \vdash e : t \right] \left( \langle n', \emptyset \rangle : S \right) \right) h' \) so the result holds by induction on \( e \) provided \( P' \cap A(n') = \text{privs}(\langle n', \emptyset \rangle : S) \). But this equality holds by Fact[7.1]

- Case \( \text{dopriv} \ p \ in \ e \): The result holds by induction for \( e \), provided that \( P \cup_n \{ p \} = \text{privs}(\langle n, P' \cup \{ p \} \rangle : S) \). This holds because for any \( p' \)

\[
\begin{align*}
p' & \in P \cup_n \{ p \} \\
\iff & \ p' \in P \cup (p' \in A(n) \land p' = p) \\
\iff & \ \text{chk}(p', \langle n, P' \rangle : S) \lor (p' \in A(n) \land p' = p) \quad \text{by } \text{def } \cup_n \\
\iff & \ (p' \in A(n) \land (p' \in P' \lor \text{chk}(p', S)) \lor (p' \in A(n) \land p' = p) \quad \text{def } \text{chk} \\
\iff & \ p' \in A(n) \land (p' \in P' \cup \{ p \} \lor \text{chk}(p', S)) \quad \text{logic and sets} \\
\iff & \ p' \in \text{privs}(\langle n, P' \cup \{ p \} \rangle : S) \quad \text{def } \text{chk and } \text{privs}
\end{align*}
\]

- Case \( \text{check} \ p \ for \ e \): Both semantics are conditional; the condition in one case is \( p \in P' \) and in the other case \( \text{chk}(p, S) \), and these are equivalent conditions by assumption \( P' = \text{privs}(S) \) for the Lemma. In case the condition is true, the result holds by induction, which applies because for both semantics the security arguments for \( e \) are unchanged. If the condition is false, the result holds because both semantics are \( \ast \) and \( \text{sim} \ t \ast \ast \).

- Case \( \text{test } p \ then \ e_1 \ else \ e_2 \): Similar to the case for \( \text{check} \).

8 Conclusion

Our work was motivated by the hope, inspired by discussions with Dave Schmidt, for more principled semantics of static analyses presented in the form of type and effect systems. Our work serves to demonstrate two attractive features of denotational semantics, which a decade ago seemed largely eclipsed by operational semantics. The first is proof of program equalities via equational reasoning on denotations. The second is logical relations, defined by structural recursion on types. We are glad for the opportunity to demonstrate the utility of denotational semantics while celebrating the contributions of Dave Schmidt, so adopt a practitioner of all forms of semantic modeling.

References

[1] M. Abadi, M. Burrows, B. W. Lampson & G. D. Plotkin (1993): *A Calculus for Access Control in Distributed Systems*. ACM Transactions on Programming Languages and Systems 15(4), pp. 706–734, doi[10.1145/155183.155225]

[2] M. S. Ager, O. Danvy & J. Midtgard (2005): *A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects*. Theoretical Computer Science 342, pp. 4–28, doi[10.1016/j.tcs.2005.06.008]
[3] A. Banerjee & D. A. Naumann (2001): *A Simple Semantics and Static Analysis for Java Security*. Technical Report CS Report 2001-1, Stevens Institute of Technology. Available at [http://www.cs.stevens-tech.edu/~naumann/tr2001.ps](http://www.cs.stevens-tech.edu/~naumann/tr2001.ps)

[4] A. Banerjee & D. A. Naumann (2002): *Representation Independence, Confinement and Access Control*. In: ACM Symposium on Principles of Programming Languages, pp. 166–177, doi:10.1145/503272.503289

[5] A. Banerjee & D. A. Naumann (2005): *Stack-based Access Control for Secure Information Flow*. Journal of Functional Programming 15(2), pp. 131–177, doi:10.1017/S0956796804005453

[6] F. Besson, T. Blanc, C. Fournet & A. D. Gordon (2004): *From Stack Inspection to Access Control: A Security Analysis for Libraries*. In: Computer Security Foundations Workshop (CSFW), pp. 61–75, doi:10.1109/CSFW.2004.11

[7] F. Besson, T. P. Jensen & D. Le Métayer (2001): *Model Checking Security Properties of Control Flow Graphs*. Journal of Computer Security 9(3), pp. 217–250.

[8] F. Besson, T. de Grenier de Latour & T. P. Jensen (2005): *Interfaces for Stack Inspection*. Journal of Functional Programming 15(2), pp. 179–217, doi:10.1017/S0956796804005465

[9] P. Centonze, R. J. Flynn & M. Pistoia (2007): *Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-Control Policies*. In: Computer Security Applications Conference (ACSAC), pp. 292–303, doi:10.1109/ACSAC.2007.14

[10] J. Clements & M. Felleisen (2004): *A Tail-recursive Machine with Stack Inspection*. ACM Transactions on Programming Languages and Systems 26(6), pp. 1029–1052, doi:10.1145/1034774.1034778

[11] C. Fournet & A. D. Gordon (2003): *Stack inspection: Theory and Variants*. ACM Transactions on Programming Languages and Systems 25(3), pp. 360–399, doi:10.1145/641909.641912

[12] L. Gong (1999): *Inside Java 2 Platform Security*. Addison-Wesley.

[13] B. A. LaMacchia, S. Lange, M. Lyons, R. Martin & K. T. Price (2002): *NET Framework Security*. Addison-Wesley.

[14] M. Pistoia, A. Banerjee & D. A. Naumann (2007): *Beyond Stack Inspection: A Unified Access-Control and Information-Flow Security Model*. In: 28th IEEE Symposium on Security and Privacy, pp. 149–163, doi:10.1109/SP.2007.10

[15] F. Pottier, C. Skalka & S. Smith (2005): *A Systematic Approach to Static Access Control*. ACM Transactions on Programming Languages and Systems 27(2), pp. 344–382, doi:10.1145/1057387.1057392

[16] C. Skalka (2002): *Types for Programming Language-Based Security*. Ph.D. thesis, The Johns Hopkins University.

[17] C. Skalka & S. Smith (2000): *Static Enforcement of Security with Types*. In: International Conference on Functional Programming, pp. 34–45, doi:10.1145/351240.351244

[18] C. Skalka, S. Smith & D. Van Horn (2008): *Types and Trace Effects of Higher Order Programs*. Journal of Functional Programming 18(2), pp. 179–249, doi:10.1017/S0956796807006466

[19] D. Wallach, A. Appel & E. Felten (2000): *SAFKASI: A Security Mechanism for Language-based Systems*. ACM Transactions on Software Engineering and Methodology 9(4), pp. 341–378, doi:10.1145/363516.363520