Static Enforcement of Security with Types

Christian Skalka
The Johns Hopkins University
ces@blaze.cs.jhu.edu

Scott Smith
The Johns Hopkins University
scott@cs.jhu.edu

ABSTRACT

A number of security systems for programming languages have recently appeared, including systems for enforcing some form of *access control*. The Java JDK 1.2 security architecture is one such system that is widely studied and used. While the architecture has many appealing features, access control checks are all implemented via dynamic method calls. This is a highly non-declarative form of specification which is hard to read, and which leads to additional run-time overhead. In this paper, we present a novel *security type system* that enforces the same security guarantees as Java Stack Inspection, but via a static type system with no additional run-time checks. The system allows security properties of programs to be clearly expressed within the types themselves. We also define and prove correct an inference algorithm for security types, meaning that the system has the potential to be layered on top of the existing Java architecture, without requiring new syntax.

1. INTRODUCTION

Security in language design is a rising concern due to the internet and increased portability of code. In particular, code-level access control—control over what code gets access to which recources—has become essential to protecting the local system from non-local, possibly untrusted code. The Java Security Architecture [7], found in Java JDK 1.2, is the most widely known and used system for enforcing code-level access control. This security architecture is implemented as a set of methods, executed during runtime, which are simply components of a security library. However, since access controls are enforced via ordinary method calls in the program itself, it is difficult to determine which access controls are actually in place by inspection of the code. Furthermore, the dynamic nature of security checks interferes with compiler optimizations (see [15]). Our goal is to develop a static, integrated and declarative security architecture for general-purpose programming languages.

In this paper, we develop a novel static type system for enforcing safety with respect to certain access control properties at run-time. In this system, security information is coded in the form of *security types*, and the type system properly enforces propagation of this information.

The aim is an expressive, flexible discipline for code-based access control, providing static verification that security checks are met, and allowing run-time security checks (and possible run-time security exceptions) to be avoided. In particular, we present an alternative to *stack inspection*, the security discipline at the heart of the JDK security architecture. This discipline is enforced by a run-time check which analyzes the call stack. The advantages of static over dynamic enforcement of properties is well-known in programming language design:

- Types themselves serve as concise readable specifications of program behavior;
- Lack of a class of run-time errors gives more reliable execution behavior;
- Compilers can generate better code if more is known about program behavior, via types, at compile time.

In the context of secure programming, "more reliable" directly translates to "more secure" and so is of particular concern. Also, security bugs often originate from a misunderstanding of how the security framework is to be properly utilized, and not from fundamental flaws in the framework itself. Thus, the usual benefits of static type frameworks provide a particular advantage over dynamic approaches to the enforcement of security.

1.1 Review and critique of the JDK Security Architecture

An oversimplified view of our goal is to produce a cleaner, more declarative version of the JDK 1.2 Security Architecture [7]. This architecture is implemented and widely used, so whatever shortcomings it may have, it is well-grounded in practicum. Thus, it is a strong grounding point for a formal treatment of access control, and this paper is intended as a basis for understanding the application of types to access control security. The reader familiar with the JDK security architecture will note that some of its more complex features, e.g. privilege inheritance and parameterized privileges, are not modeled in our system—but our goal here is more a solid foundation for static access control rather than a complete model of the JDK security architecture.

The Java system is fundamentally a *dynamic* security checking system in that access restrictions are all checked at run-time and not compile-time. To use the system, the programmer adds "do privileged" and "check privilege" commands to the code. A "do privileged" command adds a flag to the caller's stack frame, which is eliminated when the frame returns. When a privilege is checked via "check privilege" command, the stack frames are searched most to least recent. If a frame is encountered with the desired flag, the check succeeds. Also, all programs in JDK 1.2 come with a specified owner. This means that stack frames have owners, which may

$$
\begin{array}{ll}
p \in \mathit{Princ} & \textit{(principals)} \\
\pi \in \mathit{Privs} & \textit{(privileges)} \\
x \in \mathit{Var} & \textit{(variables)} \\
v ::= {}^{p}\lambda x.e \mid \mathbf{secfail} & \textit{(values)} \\
e ::= v \mid e\,e \mid \text{letpriv}\ \pi\ \text{in}\ e \mid & \\
\quad \text{checkpriv}\ \pi\ \text{for}\ e & \textit{(expressions)} \\
 & \\
\mathcal{P} ::= \{\pi_1, \ldots, \pi_j\} & \textit{(privilege sets)} \\
\mathcal{A} ::= \{p_1 \mapsto \mathcal{P}_1, \ldots, p_k \mapsto \mathcal{P}_k\} & \textit{(access credentials)} \\
\mathcal{S} ::= \ \mathit{nil} \mid \langle p, \mathcal{P} \rangle :: \mathcal{S} & \textit{(secstacks)}
\end{array}
$$

Figure 1: Grammar for λ_{sec}

be untrusted for some privilege; if such an owner is encountered before success, the check fails. See [6] for a concise description of stack inspection and how it may be used to enforce security properties. A simplified model of the JDK 1.2 security architecture and the stack inspection algorithm is presented below (section 2).

The Java security architecture is a solid proposal which is being applied in practice, but it has significant flaws. There is a performance penalty to pay due to the need for run-time stack inspection–though a technique called security-passing style [15] has been proposed to lessen the need to literally inspect every stack frame. However, even this solution does not address the *ad hoc* nature of the architecture; all security properties are enforced by method calls, a highly non-declarative form of specification. This makes the access control specification very difficult to read—it is all buried in the code and the implicit control flow structure of that code. Specifications that are difficult to read are easy to get wrong, and a security specification is the last thing you want to get wrong. This paper begins to explore some solutions to these problems through the use of static type systems.

The structure of our presentation is as follows: in Section 2, we formalize the Java Stack Inspection security enforcement mechanism within a small functional calculus. In Section 3, we define our new security type system which statically enforces all the properties of stack inspection, and thus obviates the need for run-time stack inspection. In Section 4 we define and prove correct a type inference algorithm, meaning that types can replace stack inspection transparently in our functional calculus. In Section 5, some related work is cited and described. We conclude in Section 6, with remarks on future work.

2. FORMALIZING JAVA STACK INSPECTION

In this section, we develop simple model of the Java JDK 1.2 security architecture, by defining a small language containing basic access control features, and by formalizing stack inspection for that language.

2.1 The language λ_{sec}

For our analysis we define the language λ_{sec}, a small functional language with constructs for ownership, access credential lists, and enabling and checking privileges. The language is a simplified model of the JDK 1.2 security architecture. As the initial security discipline, we define a stack inspection algorithm which is a formalization of Java stack inspection. We then use this formalization to prove soundness properties of our type system. Although the language model is simple, it covers significant ground, and more complex features can be added to this core. The addition of records and objects to the language, for example, can be added without serious additional challenge; the per-function security defined here translates to per-method security in an object-oriented language. Modules, on the other hand, are a greater challenge because component linking occurs at the module level and authentication of principals will be needed; adding modules is a topic of future work.

The language contains the usual lambda expressions, and also notions of privilege and function ownership. Privileges, denoted as π and ranging over the set of identifiers *Privs*, model rights such as PrintPriv, FileReadPriv, etc., that a system might specify. They are activated and checked by letpriv and checkpriv expressions—our version of the do privileged and check privilege commands—and every function has a specified owner p, which is the identity of a *principal* (a similar annotation of expressions with owner names is described in [8]).

In any given system, every principal p is authorized for a particular set of privileges π; this is expressed via the *access credentials* mapping $\mathcal{A}$, where for any principal p, $\mathcal{A}(p)$ is the finite set of valid privileges for p. We assume as given a possibly infinite set of privileges $\mathcal{P}_{\top}$ that the system draws upon– i.e., for all p, $\mathcal{A}(p) \subseteq \mathcal{P}_{\top}$. We say that a program is unsafe iff it reduces to the value $\mathbf{secfail}$; this value exists precisely to represent an operational security failure. The full grammar for language expressions and constructs used in the operational semantics are given in Figure 1, where *Princ*, *Privs* and *Var* are disjoint sets of identifiers. The language includes letpriv and checkpriv expressions, as well as ${}^{p}\lambda x.e$ which indicates a function owned by p, and $\mathbf{secfail}$ which is a state indicating a security violation occurred; this would be an exception in a language with exceptions but we simplify here and abort in the case of failure.

2.2 Language examples

Before precisely defining the stack inspection and reduction semantics, we informally illustrate the operational nature of the system with some simple examples. The behavior asserted in these examples may be rigorously verified by the operational semantics that follows. Note that for the moment we ignore function ownership and access credentials for ease of presentation. Let:

$$
\begin{aligned}
&\mathit{id} = \lambda x.\, x \\
&\mathit{lp} = \lambda f.\lambda x.\, \text{letpriv}\ \pi\ \text{in}\ f\, x \\
&\mathit{cp} = \lambda x.\, \text{checkpriv}\ \pi\ \text{for}\ x
\end{aligned}
$$

Each of these expressions are safe in a top-level environment; the only term that could cause problems is cp, but the security check is frozen by the lambda abstraction. If it is unfrozen by the application $\mathit{cp}\ \mathit{id}$ in an environment with the privilege π not enabled, then a security error will occur: $\mathit{cp}\ \mathit{id}$ reduces to $\mathbf{secfail}$. However, it is possible to "wrap" cp with lp, yielding a function which is equivalent to cp, but which can be safely applied in an environment without π enabled: for example, $(\mathit{lp}\ \mathit{cp})\ \mathit{id}$ is safe in any environment, because lp enables π before it is checked in the body of cp.

Privileges exist "on the stack" only as long as the stack frame in which they were granted exists. This means that in the expression $\text{letpriv}\ \pi\ \text{in}\ e$, π is enabled *only* for e at the time of evaluation. A somewhat strange consequence of this can be demonstrated with the following relationships, where $\mathit{fx} = \lambda f.\lambda x.f\, x$ and with $\simeq$ meaning operational equivalence:

$$
\begin{aligned}
(\text{letpriv}\ \pi\ \text{in}\ \mathit{fx}) &\not\simeq \mathit{lp} \\
(\text{letpriv}\ \pi\ \text{in}\ \mathit{fx}) &\simeq \mathit{fx}
\end{aligned}
$$

These relationships hold because the letpriv is frozen by lambda abstraction in lp, but not in $(\text{letpriv}\ \pi\ \text{in}\ \mathit{fx})$. As a demonstration, observe that the expression $((\text{letpriv}\ \pi\ \text{in}\ \mathit{fx})\ \mathit{cp})\ \mathit{id}$ is not safe.

$$\textit{(letpriv)} \quad \frac{\langle p, \mathcal{P} \oplus \pi \rangle :: \mathcal{S}, \mathcal{A} \vdash e \to v}{\langle p, \mathcal{P} \rangle :: \mathcal{S}, \mathcal{A} \vdash \text{letpriv } \pi \text{ in } e \to v}$$

$$\textit{(checkpriv)} \quad \frac{\text{inspect}(\mathcal{S}, \pi, \mathcal{A}) = \textit{true} \quad \mathcal{S}, \mathcal{A} \vdash e \to v}{\mathcal{S}, \mathcal{A} \vdash \text{checkpriv } \pi \text{ for } e \to v}$$

$$\textit{(appl)} \quad \frac{\mathcal{S}, \mathcal{A} \vdash e_1 \to {}^{p}\lambda x.e \quad \mathcal{S}, \mathcal{A} \vdash e_2 \to v' \quad \langle p, \{\} \rangle :: \mathcal{S}, \mathcal{A} \vdash e[v'/x] \to v}{\mathcal{S}, \mathcal{A} \vdash e_1\, e_2 \to v}$$

$$\textit{(val)} \quad \frac{}{\mathcal{S}, \mathcal{A} \vdash v \to v}$$

$$\textit{(checkprivf)} \quad \frac{\text{inspect}(\mathcal{S}, \pi, \mathcal{A}) = f}{\mathcal{S}, \mathcal{A} \vdash \text{checkpriv } \pi \text{ for } e \to \textbf{secfail}}$$

$$\textit{(applfr)} \quad \frac{\mathcal{S}, \mathcal{A} \vdash e_2 \to \textbf{secfail}}{\mathcal{S}, \mathcal{A} \vdash e_1\, e_2 \to \textbf{secfail}}$$

$$\textit{(applfl)} \quad \frac{\mathcal{S}, \mathcal{A} \vdash e_1 \to \textbf{secfail} \quad \mathcal{S}, \mathcal{A} \vdash e_2 \to v}{\mathcal{S}, \mathcal{A} \vdash e_1\, e_2 \to \textbf{secfail}}$$

Figure 2: Operational semantics for λ_{sec}

Now, we introduce the notion of function ownership. Let *cp* be redefined as follows, in the full λ_{sec} syntax:

$$cp = {}^{p}\lambda x.\text{checkpriv } \pi \text{ for } x$$

This definition specifies that the owner of *cp* is the principal p. In conjunction with ownership, the access credential list $\mathcal{A}$ becomes relevant to the safety of the language. For example, suppose that the local access control list $\mathcal{A}_1$ specifies that p is completely untrusted– that is, that p has no privileges:

$$\mathcal{A}_1 = \{\cdots, p \mapsto \emptyset, \cdots\}$$

Then the function *cp* is useless locally, since the checkpriv expression in it will fail stack inspection in every security context– the operational semantics require that the owner of a function be entitled to the privileges necessary for its use. Informally, we may say that *cp* is useless because it needs π, which p is not authorized for. Further, if *cp* were defined to call a function which needs π, then it would also fail. On the other hand, if the local access control list is $\mathcal{A}_2$, which entitles p to the privilege π:

$$\mathcal{A}_2 = \{\cdots, p \mapsto \{\pi\}, \cdots\}$$

then *cp* could safely be applied in certain environments– in particular, environments with π enabled by a preceding letpriv. In an environment of mixed local and "outsider" code, this system will prevent outside code from accessing unauthorized local resources.

For example, suppose in the local system we wish to make printing a privileged resource, available to some trusted non-local parties. In this case, where we assume the *Print* function is available only to the local system, a "secured" print function *safePrint* could be made available to external code:

$$\textit{safePrint} = {}^{\text{system}}\lambda x.\text{checkpriv PrintPriv for } \textit{Print}(x)$$

Then, code owned by some outsider p could try to use *safePrint*:

$$\textit{foreignProg} = {}^{p}\lambda x.\ldots \textit{safePrint}(\textit{text})\ldots$$

Letting $\pi = \textit{PrintPriv}$, we see that if $\mathcal{A}_1$ as defined above is the local access credential list, implying that p is not trusted for $\textit{PrintPriv}$, then *foreignProg* is useless. But if $\mathcal{A}_2$ defines the local policy, *foreignProg* can be used in certain contexts (i.e., with $\textit{PrintPriv}$ enabled).

2.3 The stack inspection algorithm

We now give the formal definition of stack inspection, which makes rigorous the informal assertions in the previous examples.

Security stacks $\mathcal{S}$ are used in the operational semantics to dynamically check security properties. Each element of the stack represents the security information for a given stack frame, notated $\langle p, \mathcal{P} \rangle$, which indicates a stack frame owned by p and containing activated privileges $\mathcal{P}$. The inspect function inspects the stack and returns whether the stack is legal or not:

$$\begin{aligned}
&\text{inspect}(\textit{nil}, \pi, \mathcal{A}) = \textit{false}\\
&\text{inspect}(\langle p, \mathcal{P} \rangle :: \mathcal{S}, \pi, \mathcal{A}) =\\
&\qquad \text{if } \pi \notin \mathcal{A}(p) \text{ then } \textit{false} \text{ else}\\
&\qquad \text{if } \pi \in \mathcal{P} \text{ then } \textit{true} \text{ else inspect}(\mathcal{S}, \pi, \mathcal{A})
\end{aligned}$$

This algorithm implements the Java stack inspection algorithm: given a privilege π, the stack is searched frame by frame from the current frame until the privilege is found on the stack (return *true*), or the owner of the frame lacks that credential (return *false*), or we ran off the top of the stack (return *false*). The set of privileges which are enabled on a particular stack $\mathcal{S}$, given some access credential list $\mathcal{A}$, is denoted privs$(\mathcal{S})$; *i.e.*,

$$\text{privs}(\mathcal{S}, \mathcal{A}) = \{\pi \mid \text{inspect}(\mathcal{S}, \pi, \mathcal{A}) = \textit{true}\}\,.$$

Using the stack inspection algorithm, we can define the operational semantics of λ_{sec}. The key differences from a standard operational semantics are that

- function application adds a new frame to the security stack,
- letpriv adds a privilege to the top frame on the stack,

$$
\begin{array}{ll}
t ::= t_1 \mid t_2 \mid \cdots & \textit{(type variables)} \\
\rho ::= \rho_1 \mid \rho_2 \mid \cdots & \textit{(privstruct variables)} \\
\Pi ::= \mathcal{P} \mid \Pi \sqcup \Pi \mid \Pi \ominus \mathcal{P} \mid \rho & \textit{(privstructs)} \\
\tau ::= t \mid \tau \xrightarrow{\Pi} \tau & \textit{(types)} \\
PC ::= & \\
\quad \{\Pi_1 \sqsubseteq \Pi'_1, \ldots, \Pi_n \sqsubseteq \Pi'_n\} & \textit{(privstruct constraints)} \\
TC ::= \{\tau_1 <: \tau'_1, \ldots, \tau_n <: \tau'_n\} & \textit{(type constraints)} \\
\Gamma ::= \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} & \textit{(type mappings)} \\
\varsigma ::= \Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau \,/\, PC, TC & \textit{(type judgements)}
\end{array}
$$

Figure 3: Security Type Grammar

- checkpriv inspects the stack via inspect to make sure the needed privilege π is found,
- if a needed privilege is not found by checkpriv, the entire computation aborts with result **secfail**.

A judgment in the operational semantics is written $\mathcal{S}, \mathcal{A} \vdash e \rightarrow v$, meaning e reduces to value v in environment $\mathcal{S}, \mathcal{A}$. The rules for the operational semantics are given in Figure 2. Note that the *letpriv* reduction rule requires that any stack $\mathcal{S}$ used in a judgement be non-empty. Since elements can only be added to the stack, this means that any evaluation must be initiated with a non-empty stack. This captures the notion of a top-level owner of the program. In our model, we assume some top-level user p_ι with $\mathcal{A}(p_\iota) = \mathcal{P}_\iota$, and define the stack $\mathcal{S}_\iota = \langle p_\iota, \mathcal{P}_\iota \rangle :: nil$ as the initial evaluation stack. The *top-level* environment is then $\mathcal{S}_\iota, \mathcal{A}$, and if $\mathcal{S}_\iota, \mathcal{A} \vdash e \rightarrow v$ we write $e \rightarrow v$ for brevity.

With this formalization, we can give a precise account of the examples in subsection 2.2; e.g. with arbitrary $\mathcal{A}$, we deduce the following using *checkprivf*, *appl* and *val*:

$$\frac{\text{inspect}(\langle p, \{\}\rangle :: \mathcal{S}_\iota, \pi, \mathcal{A}) = f}{\langle p, \{\}\rangle :: \mathcal{S}_\iota, \mathcal{A} \vdash \text{checkpriv } \pi \text{ for } id \rightarrow \textbf{secfail}}$$

$$\frac{\mathcal{S}_\iota, \mathcal{A} \vdash id \rightarrow id \quad \mathcal{S}_\iota, \mathcal{A} \vdash cp \rightarrow cp}{\mathcal{S}_\iota, \mathcal{A} \vdash cp(id) \rightarrow \textbf{secfail}}$$

3. SECURITY TYPES FOR λ_{sec}

In this section we present our type system for λ_{sec}, and establish a type safety result for the system via subject reduction. Type safety in this context means that no well-typed programs will ever have any stack inspection failures during run-time execution; we call this property *security stack safety*. This then obviates the need to perform stack inspection at run-time. The type system serves an important additional purpose: security information for programs is clearly advertised in their types, rather than being buried in function calls which the programmer must discover either through security failure, or painstaking analysis of source code.

We formulate the types here as a *constrained type system*; for background on these systems the reader is referred to any of [1, 5, 9]. Our presentation here, especially our proof method, most closely follows Henglein's presentation in [9]. The question of whether to use a constrained or constraint-free system as a basis for this work is largely one of style; we chose the former mainly because of our previous experience with such formalisms [5]. The general ideas should be applicable across a wide range of type systems. The security type grammar is defined in figure 3.

3.1 Security types overview

The most novel feature of security types are the *privstructs* Π that decorate function types. The general idea is that if a function f has type $\tau \xrightarrow{\Pi} \tau$, then Π expresses the privileges necessary to execute f. A simplified example is:

$$^{p}\lambda x.\text{checkpriv } \pi \text{ for } x \;\; : \;\; t \xrightarrow{\{\pi\}} t$$

Note that privstruct expressions Π, defined in Figure 3, include ground privilege sets $\mathcal{P}$, privilege set variables ρ, and operators thereupon.

A security type judgement is of the form

$$\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau \,/\, PC, TC$$

This judgement asserts that under type environment Γ (a mapping from expression variables to types), with Π representing the currently enabled privileges, then the code e, owned by principal p, has type τ with constraints PC, TC. PC contains the set constraints that are imposed by function ownership; for any subterm $^{p'}\lambda x.e'$ of e that needs Π', the constraint $\Pi' \sqsubseteq \mathcal{A}(p')$ is in PC. This ensures that the security requirements of functions do not exceed the privileges granted to the owner. TC is the usual type constraint set, containing elements of the form $\tau_1 <: \tau_2$.

3.2 Security type examples

In this subsection, we develop an understanding of our type system by introducing its features piece by piece. Each example is really a simplified type, with some formal features left out for the sake of clarity. Let id be defined as follows:

$$id = {}^{p}\lambda x.x$$

Then we may give id the type $\tau_{id} = t \xrightarrow{\{\}} t$, noting that it requires no privileges to execute. Such a valid type judgment with regard to a particular access control list $\mathcal{A}$ can be written as:

$$\vdash_{\mathcal{A}} id : t \xrightarrow{\{\}} t$$

Privileges start showing up in types when letpriv's and checkpriv's are used. When privileges are enabled or checked, either action is made with respect to the current security environent. For example, a privilege may be safely checked if it is in the current environment:

$$\{\pi\} \vdash_{\mathcal{A}} \text{checkpriv } \pi \text{ for } id : \tau_{id}$$

But no privileges need be in the current environment if the checkpriv is nested within a valid letpriv for the same privilege:

$$\{\} \vdash_{\mathcal{A}} \text{letpriv } \pi \text{ in checkpriv } \pi \text{ for } id : \tau_{id}$$

These ideas are formalized in the *letpriv* and *checkpriv* typing rules in figure 5. The current needs Π of an expression are generally defined by the set of un-enabled privileges that are checked in the expression.

Function application requires special consideration in the type system, since if a function f applies another function g, then the needs of g must be considered in determining the needs of f. Let f be defined as follows:

$$f = {}^{p}\lambda g.g(id)$$

In the body of f, g is applied, but the needs of g are abstract. This can be reflected in the type, through the use of a set variable ρ:

$$\vdash_{\mathcal{A}} f : (\tau_{id} \xrightarrow{\rho} t') \xrightarrow{\rho} t' \;/\; \{\rho \sqsubseteq \mathcal{A}(p)\}$$

(coerce <:) $$\frac{}{TC \cup \{\tau <: \tau'\} \vdash \tau <: \tau'}$$

(ref<:) $$\frac{}{TC \vdash \tau <: \tau}$$

(trans<:) $$\frac{TC \vdash \tau_1 <: \tau_2 \qquad TC \vdash \tau_2 <: \tau_3}{TC \vdash \tau_1 <: \tau_3}$$

(→<:) $$\frac{TC \vdash \tau_1' <: \tau_1 \qquad TC \vdash \tau_2 <: \tau_2' \qquad \Pi \subseteq \Pi'}{TC \vdash \tau_1 \xrightarrow{\Pi} \tau_2 <: \tau_1' \xrightarrow{\Pi'} \tau_2'}$$

Figure 4: Subtype Judgement Rules

Note that to the right of the $/$ symbol is a constraint set containing $\rho \sqsubseteq \mathcal{A}(p)$; this imposes the requirement that whatever ρ is, it must be a subset of the privileges entitled to p according to $\mathcal{A}$. So if we assume that $\pi \in \mathcal{A}(p)$, then ρ can be instantiated to $\{\pi\}$, and t' can be instantiated to τ_{id} to yield:

$$\vdash_{\mathcal{A}} f : (\tau_{id} \xrightarrow{\{\pi\}} \tau_{id}) \xrightarrow{\{\pi\}} \tau_{id}$$

And with *cp* defined as in subsection 2.2:

$$\vdash_{\mathcal{A}} cp : \tau_{id} \xrightarrow{\{\pi\}} \tau_{id}$$

Therefore, the application $f(cp)$ can be typed, as long as the current security environment contains π, since f needs π. In general, all types have a current security environment Π and current owner p, with the requirement that $\Pi \sqsubseteq \mathcal{A}(p)$ is satisfiable in addition to the other constraints; this Π and p are written to the left of the $\vdash_{\mathcal{A}}$ in any type judgement. So for example:

$$\{\pi\}, p' \vdash_{\mathcal{A}} f(cp) : \tau_{id}$$

where we stipulate that $\{\pi\} \subseteq \mathcal{A}(p')$ must hold for the judgement to be valid.

We now give a formal treatment of *satisfying* privstruct substitutions, which are substitutions that induce a set interpretation of privstructs associated with a type, which is consistent with the set constraints in the type.

3.3 Satisfiability and type validity

Since privstructs Π may contain set variables ρ, they are not sets *per se*, but rather datatypes that are mapped to privilege sets via substitution and a set interpretation function. For example, if we take $\Pi = \{\pi\} \sqcup \rho$, then by applying the substitution $\{\{\pi'\}/\rho\}$ to Π we obtain the privstruct $\{\pi\} \sqcup \{\pi'\}$, which is mapped to the set $\{\pi, \pi'\}$ under our set interpretation. In general, the privstruct datatype constructors $\sqcup$ and $\ominus$ are interpreted as the set operations $\cup$ and $-$. We state this formally as follows:

DEFINITION 3.1. A *substitution* is a set S of the form $\{\mathcal{P}_1/\rho_1, \ldots, \mathcal{P}_n/\rho_n\}$, with substitution application $S(\Pi)$ defined in the obvious manner. The privstruct meaning function is defined as follows:

$$\begin{array}{l} [\![\mathcal{P}]\!] = \mathcal{P} \\ [\![\Pi_1 \sqcup \Pi_2]\!] = [\![\Pi_1]\!] \cup [\![\Pi_2]\!] \\ [\![\Pi \ominus \mathcal{P}]\!] = [\![\Pi]\!] - [\![\mathcal{P}]\!] \\ [\![\rho]\!] \text{ is undefined} \end{array}$$

If $[\![S\Pi]\!]$ is defined, we say that $\Pi \in Dom(S)$. The usual set relations may be applied to sets $[\![S\Pi]\!]$, e.g. $[\![S\Pi]\!] \subseteq [\![S'\Pi']\!]$. For brevity, we say that $\Pi \subseteq \Pi'$ iff for all S with $\Pi, \Pi' \in Dom(S)$, $[\![S\Pi]\!] \subseteq [\![S\Pi']\!]$, and $\pi \in \Pi$ iff $\pi \in [\![S\Pi]\!]$ for all S for which $\Pi \in Dom(S)$. We also define an ordering $\leq$ on substitutions, where $S_1 \leq S_2$ iff $[\![S_1\Pi]\!] \subseteq [\![S_2\Pi]\!]$ for all Π such that $\Pi \in Dom(S_1) \cap Dom(S_2)$.

As described in the previous subsection, the set PC in a type judgement defines constraints on privstructs imposed by function ownership. That is, if a function f is owned by p, then the privileges required to execute f must be a subset of the privileges $\mathcal{A}(p)$ which p is entitled to. Thus, a substitution that satisfies PC is a substitution that maps all privstructs in PC to "ground" privstructs such that the constraints in PC hold under our set interpretation. In other words, S satisfies PC iff for all $\Pi_1 \sqsubset \Pi_2 \in PC$, $[\![S\Pi_1]\!] \subseteq [\![S\Pi_2]\!]$.

In addition to PC, the type constraint set TC can also include privstruct constraints. Security types include a subtyping judgement $TC \vdash \tau_1 <: \tau_2$, defined in figure 4. We let $TC \vdash TC'$ abbreviate $TC \vdash \tau <: \tau'$ for all $\tau <: \tau' \in TC'$. Note that $\vdash$ is transitive over constraint sets:

PROPOSITION 3.2. If $TC \vdash TC'$ and $TC' \vdash TC''$, then $TC \vdash TC''$.

PROOF. By induction on the derivation of $TC' \vdash \tau <: \tau'$, where $\tau <: \tau' \in TC''$. $\square$

Thus, the subtyping relations in any TC impose constraints on privstructs because in order for the type system to be sound, $TC \vdash \tau_1 \xrightarrow{\Pi} \tau_2 <: \tau_1' \xrightarrow{\Pi'} \tau_2' \in TC$ must imply $\Pi \subseteq \Pi'$. Intuitively, this is because it is safe to overestimate the security requirements of a function, but it is not safe to underestimate them.

Privstruct constraint satisfaction is defined via a standard closure operation which is the transitive and functional closure of type constraint sets TC. First, we give a definition of closure for any set TC:

DEFINITION 3.3 (TC CLOSURE). The closure of TC, denoted $close(TC)$, is the least set TC' satisfying the following properties:

- $TC \subseteq TC'$
- If $\tau_1 \xrightarrow{\Pi} \tau_2 <: \tau_1' \xrightarrow{\Pi'} \tau_2' \in TC'$, then $\tau_1' <: \tau_1 \in TC'$ and $\tau_2 <: \tau_2' \in TC'$ (*fn* closure).
- If $\tau_1 <: \tau_2 \in TC'$ and $\tau_2 <: \tau_3 \in TC'$, then $\tau_1 <: \tau_3 \in TC'$. (transitive closure)

$close(TC)$ is easily shown to be computable by iteration.

Now, satisfiability of set, type constraint pairs PC, TC may be defined.

DEFINITION 3.4 (PRIVSTRUCT SATISFIABILITY). Let

$$scs(TC) = \left\{ \Pi \sqsubseteq \Pi' \mid \tau_1 \xrightarrow{\Pi} \tau_2 <: \tau_1' \xrightarrow{\Pi'} \tau_2' \in TC \right\}$$

and let $TPC = scs(close(TC))$. Then the set, type constraint pair PC, TC is *satisfiable* iff there exists some S such that for all $\Pi_1 \sqsubseteq \Pi_2 \in PC \cup TPC$ it is the case that $[\![S\Pi_1]\!] \subseteq [\![S\Pi_2]\!]$. The function name *scs* stands for "set constraints", because it gleans set constraints imposed by the type coercions in TC.

$$\textit{(var)} \quad \frac{\Gamma(x) = \tau}{\Gamma, \Pi, p \ \vdash_{\mathcal{A}} x : \tau \ / \ PC, TC}$$

$$\textit{(letpriv)} \quad \frac{\Gamma, \Pi', p \vdash_{\mathcal{A}} e : \tau \ / \ PC, TC}{\Gamma, \Pi, p \vdash_{\mathcal{A}} \text{letpriv } \pi \text{ in } e : \tau \ / \ PC, TC} \qquad \Pi' = \begin{cases} \Pi \sqcup \{\pi\} \text{ if } \pi \in \mathcal{A}(p) \\ \Pi \text{ otherwise} \end{cases}$$

$$\textit{(checkpriv)} \quad \frac{\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau \ / \ PC, TC \qquad \pi \in \Pi}{\Gamma, \Pi, p \vdash_{\mathcal{A}} \text{checkpriv } \pi \text{ for } e : \tau \ / \ PC, TC}$$

$$\textit{(fn)} \quad \frac{\Gamma \oplus x \mapsto \tau, \Pi', p' \ \vdash_{\mathcal{A}} e : \tau' \ / \ PC, TC \qquad \Pi' \sqsubseteq \mathcal{A}(p') \in PC}{\Gamma, \Pi, p \vdash_{\mathcal{A}} {}^{p'}\lambda x.e : \tau \xrightarrow{\Pi'} \tau' \ / \ PC, TC}$$

$$\textit{(appl)} \quad \frac{\Gamma, \Pi, p \vdash_{\mathcal{A}} e_1 : \tau' \xrightarrow{\Pi'} \tau \ / \ PC, TC \qquad \Gamma, \Pi, p \vdash_{\mathcal{A}} e_2 : \tau' \ / \ PC, TC \qquad \Pi' \subseteq \Pi}{\Gamma, \Pi, p \vdash_{\mathcal{A}} e_1 \, e_2 : \tau \ / \ PC, TC}$$

$$\textit{(sub)} \quad \frac{\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau' \ / \ PC, TC \qquad TC \vdash \tau' <: \tau}{\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau \ / \ PC, TC}$$

Figure 5: Type Judgement Rules

The above definition of *satisfiable* is elegant, but needs a stronger characterization, *csatisfiable*, for use in subject reduction.

DEFINITION 3.5 (TC-CONSISTENCY). The type constraint set TC is *consistent* iff for all derivations $TC \vdash \tau_1 \xrightarrow{\Pi} \tau_2 <: \tau_1' \xrightarrow{\Pi'} \tau_2'$, there exists a derivation with the same conclusion and with the final step an instance of $\to<:$.

DEFINITION 3.6 (SATISFIABILITY CHARACTERIZATION). The set, type constraint set pair PC, TC is *csatisfiable* iff there exists some S such that for all $\Pi_1 \sqsubseteq \Pi_2 \in PC$ it is the case that $[\![S\Pi_1]\!] \subseteq [\![S\Pi_2]\!]$, and there exists a consistent TC' such that $TC' \vdash S(TC)$. Abusing terminology, we may also speak of S satisfying PC or TC alone, with the obvious meaning.

The following lemma establishes the interchangeability of satisfiability and its characterization:

LEMMA 3.7. S satisfies PC, TC iff S csatisfies PC, TC. $\square$

Subtyping judgements are closed under satisfying substitutions, a fact demonstrated by the following proposition:

PROPOSITION 3.8. If S satisfies TC and $TC \vdash \tau <: \tau'$, then $S(TC) \vdash S\tau <: S\tau'$.

PROOF. By rule induction on $TC \vdash \tau <: \tau'$. $\square$

With this notion of constraint satisfiability, we can formally define type validity as follows:

DEFINITION 3.9 (TYPE VALIDITY). The typing $\Gamma, \Pi, \vdash_{\mathcal{A}} e : \tau \ / \ PC, TC$ is *valid* iff there exists a derivation with this conclusion, and an S that satisfies $PC \cup \{\Pi \sqsubseteq \mathcal{A}(p)\}, TC$.

If $\{\}, \Pi, p_\iota \vdash_{\mathcal{A}} e : \tau \ / \ TC, PC$ and $PC' = PC \cup \{\Pi \subseteq \mathcal{A}(p_\iota)\}$ then we write $\vdash_{\mathcal{A}} e : \tau \ / \ PC', TC$ for brevity, and we write $e : \tau$ if $\mathcal{A}$ is understood from context and PC, TC and Π are empty. If a type contains no set variables ρ, we say that it is *concrete*; the following result establishes that if e has a type, then it has a concrete type:

PROPOSITION 3.10. If $\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau \ / \ PC, TC$ is satisfied by S, then $S\Gamma, S\Pi, p \vdash_{\mathcal{A}} e : S\tau \ / \ S(PC), S(TC)$ which is concrete.

PROOF. Since S generates a concrete type by definition, the result is implied by the validity of the generated type, which follows by induction on the length of the type derivation. $\square$

3.4 Type safety and subject reduction

Our goal for security types is to prove that any well-typed program is operationally safe according to the Java stack inspection model defined in section 1. To characterize the desired theorem, we use the notion of security failure specified in the operational semantics: any unsafe program reduces to **secfail**.

THEOREM 3.11 (SECURITY TYPE SAFETY). If the type $\vdash_{\mathcal{A}} e : \tau \ / \ PC, TC$ is valid, then it is not the case that $\mathcal{S}_\iota, \mathcal{A} \vdash e \to$ **secfail**.

The theorem is proved by a standard subject reduction argument. Note that we phrase subject reduction in terms of concrete types, since operational judgements are always concrete in the sense that privs$(\mathcal{S}, \mathcal{A})$ is always a ground set $\mathcal{P}$ for any stack $\mathcal{S}$.

THEOREM 3.12 (SUBJECT REDUCTION). If $\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau \ / \ PC, TC$ is concrete, and $\mathcal{S}, \mathcal{A} \vdash e \to v$ with $[\![\Pi]\!] \subseteq$ privs$(\mathcal{S}, \mathcal{A})$ and p in the head of $\mathcal{S}$, then $\Gamma, \Pi, p \vdash_{\mathcal{A}} v : \tau \ / \ PC, TC$.

To prove subject reduction, we first demonstrate two lemmas related to subtyping which allow us to deal almost exclusively with the non-subsumption rules in the induction for subject reduction; we also demonstrate the usual "substitution" lemma in a form appropriate for security types.

LEMMA 3.13. If $\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau \ / \ PC, TC$, then there exists a subderivation of $\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau' \ / \ PC, TC$, where $TC \vdash \tau' <: \tau$, such that if $e \equiv x$, then the last step in the subderivation is an instance of *var*, if $e \equiv {}^{p}\lambda x.e$ then the last step in the subderivation is an instance of *fn*, *etc.* for each term type and the corresponding type rule.

PROOF. Each term form clearly must have an instance of its corresponding type rule in the type derivation. And since only instances of *sub* can be interposed between the derivations of $\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau' \,/\, PC, TC$ and $\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau \,/\, PC, TC$, the lemma follows via a straightforward induction on the number of interposed steps. □

LEMMA 3.14 (SUBSTITUTION). Supposing $TC \vdash \tau_1' <: \tau_1$, if $\Gamma \oplus x \mapsto \tau_1', \Pi, p \vdash_{\mathcal{A}} e : \tau_2 \,/\, PC, TC$ and $\Gamma, \Pi', p' \vdash_{\mathcal{A}} v : \tau_1 \,/\, PC, TC$, then $\Gamma, \Pi, p, \vdash_{\mathcal{A}} e[v/x] : \tau_2 \,/\, PC, TC$.

PROOF. By structural induction on e and case analysis. Here we give only the application case, letting $\Gamma' = \Gamma \oplus x \mapsto \tau_1'$:

$(e \equiv e_1 e_2)$ By lemma 3.13, there exists the following step in the derivation of the type of e in this case, where $\tau_r <: \tau_2$:

$$\frac{\begin{array}{c}\Gamma', \Pi, p \vdash_{\mathcal{A}} e_1 : \tau_d \xrightarrow{\Pi_{dr}} \tau_r \,/\, PC, TC \\ \Gamma', \Pi, p \vdash_{\mathcal{A}} e_2 : \tau_d \,/\, PC, TC \\ \Pi_{dr} \subseteq \Pi\end{array}}{\Gamma', \Pi, p \vdash_{\mathcal{A}} e_1 e_2 : \tau_r \,/\, PC, TC}$$

By the induction hypothesis, $\Gamma, \Pi, p \vdash_{\mathcal{A}} e_1[v/x] : \tau_d \xrightarrow{\Pi_{dr}} \tau_r \,/\, PC, TC$ and $\Gamma, \Pi, p \vdash_{\mathcal{A}} e_2[v/x] : \tau_d \,/\, PC, TC$. Thus, $\Gamma, \Pi, p \vdash_{\mathcal{A}} e[v/x] : \tau_r \,/\, PC, TC$ by the definition of substitution and *appl*, so the lemma holds in this case by *sub*. □

The following is another "utility" lemma for subject reduction, showing that a value is safe in any current security context:

LEMMA 3.15. If $\Gamma, \Pi, p \vdash v : \tau \,/\, PC, TC$ and $TC \vdash \tau <: \tau'$, then $\Gamma, \Pi', p' \vdash v : \tau' \,/\, PC, TC$. □

Now, we can prove the necessary subject reduction result:

PROOF OF THEOREM 3.12 By induction on the derivation of $\mathcal{S}, \mathcal{A} \vdash e \to v$. Here we show only the appication case: $(e \equiv e_1 e_2)$ By lemma 3.13, there exists the following step in the derivation of the type of e in this case, where $TC \vdash \tau_r <: \tau$:

$$\frac{\begin{array}{c}\Gamma, \Pi, p \vdash_{\mathcal{A}} e_1 : \tau_d \xrightarrow{\Pi_{dr}} \tau_r \,/\, PC, TC \\ \Gamma, \Pi, p \vdash_{\mathcal{A}} e_2 : \tau_d \,/\, PC, TC \\ \Pi_{dr} \subseteq \Pi\end{array}}{\Gamma, \Pi, p \vdash_{\mathcal{A}} e_1 e_2 : \tau_r \,/\, PC, TC}$$

By the operational semantics in this case, $\mathcal{S}, \mathcal{A} \vdash e_1 \to {}^{p'}\lambda x.e'$ and $\mathcal{S}, \mathcal{A} \vdash ve_2 \to v'$, and $\langle p', \{\}\rangle :: \mathcal{S}, \mathcal{A} \vdash e[v/x] \to v$. Thus, by the induction hypothesis, $\Gamma, \Pi, p \vdash_{\mathcal{A}} {}^{p'}\lambda x.e' : \tau_d \xrightarrow{\Pi_{dr}} \tau_r \,/\, PC, TC$ and $\Gamma, \Pi, p \vdash_{\mathcal{A}} v' : \tau_d \,/\, PC, TC$. Now, by lemma 3.13 there exists the following step in the derivation of the type of ${}^{p'}\lambda x.e'$:

$$\frac{\begin{array}{c}\Gamma \oplus x \mapsto \tau_d', \Pi_{dr}', p' \vdash_{\mathcal{A}} e' : \tau_r' \,/\, C \\ \Pi_{dr}' \sqsubseteq \mathcal{A}(p') \in PC, TC\end{array}}{\Gamma, \Pi, p \vdash_{\mathcal{A}} {}^{p'}\lambda x.e' : \tau_d' \xrightarrow{\Pi_{dr}'} \tau_r' \,/\, PC, TC}$$

where $TC \vdash \tau_d <: \tau_d'$, $TC \vdash \tau_r' <: \tau_r$ and $[\![\Pi_{dr}']\!] \subseteq [\![\Pi_{dr}]\!]$ by the concreteness of PC, TC and satisfiability. But $\Gamma \oplus x \mapsto \tau_d', \Pi_{dr}', p' \vdash_{\mathcal{A}} e' : \tau_r \,/\, PC, TC$ by *sub*, so that $\Gamma, \Pi_{dr}', p' \vdash_{\mathcal{A}} e[v'/x] : \tau_r \,/\, PC, TC$ by lemma 3.14 and *sub*. And since $\Pi_{dr} \subseteq \Pi$ and $[\![\Pi_{dr}']\!] \subseteq [\![\Pi_{dr}]\!]$ therefore $[\![\Pi_{dr}']\!] \subseteq [\![\Pi]\!]$. Further, $\Pi_{dr}' \sqsubseteq \mathcal{A}(p') \in PC$ so that $[\![\Pi_{dr}']\!] \subseteq \mathcal{A}(p')$, since PC is concrete and satisfied; therefore $[\![\Pi_{dr}']\!] \subseteq \text{privs}(\langle p', \{\}\rangle :: \mathcal{S}, \mathcal{A})$ by the definition of privs. Thus, by the induction hypothesis $\Gamma, \Pi_{dr}', p' \vdash_{\mathcal{A}} v : \tau_r \,/\, PC, TC$, so that $\Gamma, \Pi, p \vdash_{\mathcal{A}} v : \tau \,/\, PC, TC$ in this case by *sub* and lemma 3.15. □

Having established subject reduction, it is easy to demonstrate security type safety:

PROOF OF THEOREM 3.11 By proposition 3.10, if e has a type then e has a concrete type. Therefore the theorem follows by theorem 3.12, since **secfail** has no type. □

3.5 Incompleteness of the type system

There are a few sources of inconsistency in the security type system—that is, instances in which operationally safe programs do not have a type. Some are inherent in the approach, while some reflect choices that were made in the design of the system. For example, assuming that $\pi \notin \mathcal{A}(p)$, the program:

$${}^{p}\lambda x.\text{checkpriv } \pi \text{ for } x$$

does not have a type, but is operationally safe, since the program defines but does not use the function. However, while it is possible to construct the type system so that these sorts of programs are accepted, we argue that such useless functions should not be allowed, since they are just that—useless.

There are of course the standard sources of incompleteness that cannot be avoided. For example, assuming a standard encoding of conditional expressions, application of the following function in a current environment (*i.e.*, without π activated) is not well typed, though it is always operationally safe:

$${}^{p}\lambda x.\text{if } \mathit{false} \text{ then checkpriv } \pi \text{ for } x \text{ else } x$$

Even though the checkpriv branch is never taken, this sort of property cannot in general be established by our type system. Nevertheless, this kind of incompleteness should not have a significant effect on the usefulness of security types.

A third sort of incompleteness points to the need for polymorphism, which we left out of this paper to allow us to focus on the core security issues. Assuming the standard encoding for sequencing and let expressions, let e be defined as follows:

$$\begin{array}{l}\text{let} \\ \quad \mathit{id} = {}^{p}\lambda x.x \\ \quad \mathit{cp} = {}^{p}\lambda x.\text{checkpriv } \pi \text{ for } x \\ \quad f = {}^{p}\lambda f.{}^{p}\lambda x.f(x) \\ \text{in} \\ \quad f(\mathit{cp}); f(\mathit{id})\end{array}$$

Then $e : t \xrightarrow{\{\pi\}} t$ fails, since our monomorphic type system cannot distinguish between the two different application points of f, but unifies them. This example demonstrates a need for at least let-polymorphism.

In addition to issues related to the current language, branching on privileges may be a desirable idiom for a secure language. That is, suppose testpriv is a predicate on privileges, which holds iff a checkpriv of the same privilege suceeds (note, in a language with exceptions, testpriv could be defined using checkpriv and an exception handler). Then expressions such as

$$\text{if testpriv}(\pi) \text{ then } e_1 \text{ else } e_2$$

```
$isecty(\Gamma, p, x) = (\{\}, \Gamma(x), \{\}, \{\})$
$isecty(\Gamma, p, \text{letpriv } \pi \text{ in } e) =$
        let $(\Pi, \tau, PC, TC) = isecty(\Gamma, p, e)$
        and $\Pi' = \text{if } \pi \in \mathcal{A}(p) \text{ then } \Pi \ominus \pi \text{ else } \Pi$
        in $(\Pi', \tau, PC, TC)$
$isecty(\Gamma, p, \text{checkpriv } \pi \text{ for } e) =$
        let $(\Pi, \tau, PC, TC) = isecty(\Gamma, p, e)$
        in $(\Pi \sqcup \{\pi\}, \tau, PC, TC)$
$isecty(\Gamma, p, {}^{p'}\lambda x.e) =$
        let $(\Pi, \tau, PC, TC) = isecty(\Gamma \oplus x \mapsto t, p', e)$,
            where $t$ is fresh
        in $(\{\}, t \xrightarrow{\Pi} \tau, PC \cup \{\Pi \sqsubseteq \mathcal{A}(p')\}, TC)$
$isecty(\Gamma, p, e_1 e_2) =$
        let $(\Pi_1, \tau_1, PC_1, TC_1) = isecty(\Gamma, p, e_1)$
        and $(\Pi_2, \tau_2, PC_2, TC_2) = isecty(\Gamma, p, e_2)$
        and $\Pi = \Pi_1 \sqcup \Pi_2 \sqcup \rho$
        and $TC = TC_1 \cup TC_2 \cup \left\{\tau_1 <: \tau_2 \xrightarrow{\rho} t\right\}$,
            where $t$ and $\rho$ are fresh
        in $(\Pi, t, PC_1 \cup PC_2, TC)$
```

Figure 6: The *isecty* Algorithm

could be useful, and indeed this sort of expression is found in Java JDK 1.2 programs. However, our system as currently conceived does not include types that depend on the dynamic security level of the execution context. But, there is a practical need to perform such a case analysis to allow for rollback behavior in the event some higher access right is unavailable but a weaker alternative is available. So, in future work we plan to extend our type system in this direction.

4. SECURITY TYPE INFERENCE

Some form of type inference is an essential component of static security typing, since we do not want to burden users with the need to modify most types in their programs. In the following presentation, type inference requires no input from the user so is optimal in that sense, but does not always produce easily readable types. So we do not claim the final solution has been achieved.

Our type inference algorithm, *infer_secty* defined in figure 7, resembles the standard constraint type inference algorithms (*e.g.*, see [5]), with the addition of an algorithm for inferring a satisfying substitution S for any constraint sets PC, TC generated by type inference.

As usual, the correctness of type inference is stated in terms of soundness and completeness—that is, that any inferred type ς is valid, and if a term has a type ς' then a type ς will be inferred that is "most general", in the sense that ς' is an instance of ς. We follow Henglein's syntactic *halbstark* relation, described in [9], in characterizing our instance relation. Note that the definition does not require that either typing in a relation $\varsigma \preceq \varsigma'$ be valid:

DEFINITION 4.1 (INSTANCE RELATION). The type $\varsigma' \equiv \Gamma', \Pi', p \vdash_{\mathcal{A}} e : \tau' \;/\; PC', TC'$ is an *instance* of $\varsigma \equiv \Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau \;/\; PC, TC$, written $\varsigma \preceq \varsigma'$, iff there exist Θ and $\hat{S}$ such that

1. $\Gamma' = \Theta\Gamma$
2. $TC' \vdash \hat{S}(\Theta TC)$
3. $TC' \vdash \hat{S}(\Theta\tau) <: \tau'$
4. $\hat{S}\Pi \subseteq \Pi'$ and $\hat{S}(PC') \leq PC$

In the above, Θ is a security type substitution defined in the obvious manner, and $\hat{S}$ is a privstruct substitution defined similarly as in definition 3.1, except that any ρ may be mapped to a privstruct containing other set variables (i.e., it is not a "ground" substitution). For clause 4, the relation $PC_1 \leq PC_2$ holds iff for all $\Pi_2 \sqsubseteq \mathcal{A}(p) \in PC_2$ there exists $\Pi_1 \sqsubseteq \mathcal{A}(p) \in PC_1$ such that $\Pi_1 \subseteq \Pi_2$.

For the purposes of our proofs, we must at least establish that if $\varsigma \preceq \varsigma'$ and ς is valid, then so is ς'. We do this with the following lemmas.

LEMMA 4.2. If S satisfies PC, TC and $TC \vdash S(TC')$ and $PC' \leq PC$, then S satisfies PC', TC'.

PROOF. By definition 3.4, and transitivity of $\subseteq$ and $\vdash$. $\square$

LEMMA 4.3. If ς is a valid type and $\varsigma' \preceq \varsigma$, then ς' is valid.

PROOF. By lemma 4.2 and rule induction on ς. $\square$

With this characterization of $\preceq$, our main result for type inference may be stated as follows.

THEOREM 4.4 (CORRECTNESS OF TYPE INFERENCE). The type inference algorithm *infer_secty* returns $\tau \;/\; PC, TC$ for e iff $\varsigma \equiv \vdash_{\mathcal{A}} e : \tau \;/\; PC, TC$ is a valid type judgement, and for all valid type judgements $\varsigma' \equiv \vdash_{\mathcal{A}} e : \tau' \;/\; PC', TC'$ it is the case that $\varsigma \preceq \varsigma'$.

This theorem is proven in the following subsections. Inspection of the *infer_secty* reveals that the real work of the algorithm is done by the auxiliary functions *isecty*, which infers the type structure, and *satisfy*, which verifies the existence of a satisfying substitution for the type. Hence, the correctness proof of *infer_secty* is accomplished by way of correctness proofs for *isecty* (in particular, we prove soundness and completeness results for this algorithm), and *satisfy*. First, we give a brief example to illustrate the workings of the inference algorithm.

4.1 Type inference example

Let *id* and *cp* be defined as in subsection 3.1. Suppose we are inferring the type of *cp*(*id*); then *isecty* will infer the following types for these functions separately:

$$\{\}, \{\}, p_\iota \vdash_{\mathcal{A}} cp : t_1 \xrightarrow{\{\pi\}} t_1 \;/\; \{\{\pi\} \sqsubseteq \mathcal{A}(p)\}, \{\}$$

$$\{\}, \{\}, p_\iota \vdash_{\mathcal{A}} id : t_2 \xrightarrow{\{\}} t_2 \;/\; \{\{\} \sqsubseteq \mathcal{A}(p)\}, \{\}$$

Now, let:

$$PC = \{\{\pi\} \sqsubseteq \mathcal{A}(p), \{\} \sqsubseteq \mathcal{A}(p)\}$$

$$TC = \{\ t_1 \xrightarrow{\{\pi\}} t_1 <: (t_2 \xrightarrow{\{\}} t_2) \xrightarrow{\rho} t_3\ \}$$

Then the top-level type inferred for the expression $cp(id)$ will be

$$\{\}, \{\rho\}, p_\iota \vdash_{\mathcal{A}} cp(id) : t_3 \;/\; PC, TC$$

Note that TC imposes the constraint $\{\pi\} \sqsubseteq \rho$ by definition 3.4, so that in order for this type to be valid, the constraints

$$\{\pi\} \sqsubseteq \rho,\ \rho \sqsubseteq \mathcal{A}(p_\iota),\ \{\pi\} \sqsubseteq \mathcal{A}(p),\ \{\} \sqsubseteq \mathcal{A}(p)$$

```
$\mathit{satisfy}(PC, TPC) =$
    let
      $\mathit{glb}(\mathcal{P}, rr) = \mathcal{P}$
      $\mathit{glb}(\Pi_1 \sqcup \Pi_2, rr) = \mathit{glb}(\Pi_1, rr) \cup \mathit{glb}(\Pi_2, rr)$
      $\mathit{glb}(\Pi \ominus \mathcal{P}, rr) = \mathit{glb}(\Pi_1, rr) - \mathcal{P}$
      $\mathit{glb}(\rho, rr) =$
          if $\rho \in rr$ then $\{\}$
          else $\bigcup_{\Pi \sqsubseteq \rho \in TPC} \mathit{glb}(\Pi, rr \cup \{\rho\})$
    in
      $\mathit{true}$ iff for all $\Pi \sqsubseteq \mathcal{A}(p) \in PC,\ \mathit{glb}(\Pi, \emptyset) \subseteq \mathcal{A}(p)$

$\mathit{infer\_secty}(e) =$
      let $(\Pi, \tau, PC, TC) = \mathit{isecty}(\{\}, p_\iota, e)$
      and $cTC = \mathit{close}(TC)$
      and $TPC = \mathit{scs}(cTC)$
      and $PC' = PC \cup \{\Pi \sqsubseteq \mathcal{A}(p_\iota)\}$
      in
          if $\mathit{satisfy}(PC', TPC)$ then $\tau \,/\, PC', TC$
          else raise **sectyfail**
```

Figure 7: The *infer_secty* and *satisfy* Algorithms

must all be satisfiable. Of course, this is possible only if $\pi \in \mathcal{A}(p)$ and $\pi \in \mathcal{A}(p_\iota)$, in which case *e.g.* $\{\{\pi\}/\rho\}$ satisfies these constraints. In any case, *satisfy* addresses the problem of set constraint satisfiability.

4.2 Soundness of *isecty*

The soundness result for *isecty* establishes that any type returned is indeed valid, modulo the satisfiability of the constraint sets generated; this is because the issue of satisfiability is handled not by *isecty*, but by *satisfy*. To assist in the soundness proof, we demonstrate the following lemma, which shows that type judgements are preserved by "pumping up" the constraint sets:

LEMMA 4.5. If $\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau \,/\, PC, TC$ and $\Pi \subseteq \Pi'$, $PC \subseteq PC'$ and $TC \subseteq TC'$, where $PC' \cup \{\Pi' \sqsubseteq \mathcal{A}(p)\}, TC'$ is satisfiable, then $\Gamma, \Pi', p \vdash_{\mathcal{A}} e : \tau \,/\, PC', TC'$.

PROOF. The lemma follows by definition of the typing and $<:$ rules. $\square$

Now it is easy to state and prove the appropriate soundness result:

LEMMA 4.6 (SOUNDNESS OF *isecty*). If $\mathit{isecty}(\Gamma, e, p)$ returns (Π, τ, PC, TC) and $PC \cup \Pi \sqsubseteq \mathcal{A}(p), TC$ is satisfiable, then $\Gamma, \Pi, p \vdash_{\mathcal{A}} e : \tau \,/\, PC, TC$.

PROOF. The lemma follows by structural induction on e. Here we demonstrate only the case $(e \equiv e_1 e_2)$. Let $\mathit{isecty}(\Gamma, e_1, p)$ return $(\Pi_1, \tau_1, PC_1, TC_1)$ and $\mathit{isecty}(\Gamma, e_2, p)$ return $(\Pi_2, \tau_2, PC_2, TC_2)$; then by the definition of the algorithm:

$$\begin{aligned}
&\tau = t\\
&PC = PC_1 \cup PC_2\\
&TC = TC_1 \cup TC_2 \cup \left\{\tau_1 <: \tau_2 \xrightarrow{\rho} t\right\}\\
&\Pi = \Pi_1 \sqcup \Pi_2 \sqcup \rho
\end{aligned}$$

By the induction hypothesis, $\Gamma, \Pi_1, p \vdash_{\mathcal{A}} e_1 : \tau_1 \,/\, TC_1, PC_1$ and $\Gamma, \Pi_2, p \vdash_{\mathcal{A}} e_2 : \tau_2 \,/\, TC_2, PC_2$, therefore $\Gamma, \Pi, p \vdash_{\mathcal{A}} e_2 : \tau_2 \,/\, TC, PC$ and $\Gamma, \Pi, p \vdash_{\mathcal{A}} e_1 : \tau_1 \,/\, TC, PC$ by lemma 4.5. Thus $\Gamma, \Pi, p \vdash_{\mathcal{A}} e_1 : \tau_2 \xrightarrow{\rho} t \,/\, TC, PC$ by *sub*, and clearly $\rho \subseteq \Pi$, therefore $\Gamma, \Pi, p \vdash_{\mathcal{A}} e : t \,/\, TC, PC$ by *appl*. $\square$

4.3 Completeness of *isecty*

Our *isecty* completeness result shows that the algorithm returns a most general type, in the sense formalized by the $\preceq$ relation. For the purposes of the proof we informally note that any type mapping Γ used by *isecty* maps expression variables x to type variables t, and observe that this invariant is maintained throughout execution of *isecty* by definition of the algorithm.

LEMMA 4.7 (COMPLETENESS OF *isecty*). If the type $\varsigma \equiv \Theta(\Gamma), \Pi, p \vdash_{\mathcal{A}} e : \tau \,/\, PC, TC$ is valid, then $\mathit{isecty}(\Gamma, p, e)$ returns (Π', τ', PC', TC') such that $\Gamma, \Pi', p \vdash_{\mathcal{A}} e : \tau' \,/\, PC', TC'$ is valid and $\preceq \varsigma$.

PROOF. By lemma 4.3, the proof follows by showing that the type returned by *isecty* is most general according to $\preceq$. We establish this property by induction on the type judgement; here we show only the case $(e = e_1 e_2)$. By lemma 3.13, there exists the following step in the derivation of the type of e, where $\tau_r <: \tau$:

$$\frac{\Theta\Gamma, \Pi, p \vdash_{\mathcal{A}} e_1 : \tau_d \xrightarrow{\Pi_{dr}} \tau_r \,/\, PC, TC \qquad \Theta\Gamma, \Pi, p \vdash_{\mathcal{A}} e_2 : \tau_d \,/\, PC, TC \qquad \Pi_{dr} \subseteq \Pi}{\Theta\Gamma, \Pi, p \vdash_{\mathcal{A}} e_1 e_2 : \tau_r \,/\, PC, TC}$$

Let $\mathit{isecty}(\Gamma, e_1, p)$ return $(\Pi_1, \tau_1, PC_1, TC_1)$ and let $\mathit{isecty}(\Gamma, e_2, p)$ return $(\Pi_2, \tau_2, PC_2, TC_2)$; then by the definition of the algorithm:

$$\begin{aligned}
&\tau' = t\\
&PC' = PC_1 \cup PC_2\\
&TC' = TC_1 \cup TC_2 \cup \left\{\tau_1 <: \tau_2 \xrightarrow{\rho} t\right\}\\
&\Pi' = \Pi_1 \sqcup \Pi_2 \sqcup \rho
\end{aligned}$$

By the induction hypothesis, there exist substitutions Θ_1, $\hat{S}_1$, Θ_2 and $\hat{S}_2$ such that for $i \in \{1, 2\}$ it case that $\hat{S}_i(\Pi_i) \subseteq \Pi$, $TC \vdash \hat{S}_i(\Theta_i(TC_i))$, etc. for each condition of the appropriate $\preceq$ relations. We note that in objects X and Y returned by distinct recursive calls to *isecty*, any set variables will be fresh, so that $\hat{S}_1$ and $\hat{S}_2$ have disjoint domains. Furthemore, by construction of the proof, Θ_1 and Θ_2 need differ from Θ only in terms of those type variables generated for the application case, which are fresh. Thus, there exist $\Theta' = \Theta_1 \circ \Theta_2 = \Theta_2 \circ \Theta_1$ and $\hat{S} = \hat{S}_1 \circ \hat{S}_2 = \hat{S}_2 \circ \hat{S}_1$ such that $\hat{S}(\Pi_1 \sqcup \Pi_2) \subseteq \Pi$, $\hat{S}(PC') \leq PC$ and $TC \vdash \hat{S}(\Theta'(TC_1 \cup TC_2)$, and also $TC \vdash \hat{S}(\Theta'\tau_1) <: \tau_d \xrightarrow{\Pi_{dr}} \tau_r$ and $TC \vdash \hat{S}(\Theta'\tau_2) <: \tau_d$. Let $\Theta'' = \Theta' \cup \{\tau_r/t\}$ and $\hat{S}' = \hat{S} \cup \{\Pi_{dr}/\rho\}$; since t and ρ are fresh, therefore $\hat{S}'(\Theta''(\tau_2 \xrightarrow{\rho} t)) = \hat{S}'(\Theta''\tau_2) \xrightarrow{\Pi_{dr}} \tau_r$ and $TC \vdash \hat{S}'(\Theta''\tau_2) <: \tau_d$, so that $TC \vdash \hat{S}'(\Theta''\tau_1) <: \hat{S}'(\Theta''\tau_2) \xrightarrow{\Pi_{dr}} \tau_r$ by $\rightarrow<:$. And therefore $TC \vdash \hat{S}'(\Theta'' TC')$, and clearly $\hat{S}'\Pi' \subseteq \Pi$, etc., so this case holds by *sub* and definition 4.1, since $\hat{S}'(\Theta''t) = \tau_r$ and $TC \vdash \tau_r <: \tau$. $\square$

4.4 Correctness of *satisfy* and *infer_secty*

The final step in establishing the correctness of type inference is the proof of corretness for satisfy, which we state as follows:

```
$mt_{TPC}\ (rr, S)\ \mathcal{P} = \mathcal{P}$
$mt_{TPC}\ (rr, S)\ \Pi_1 \sqcup \Pi_2 =$
        $(mt_{TPC}\ (rr, S)\ \Pi_1) \cup (mt_{TPC}\ (rr, S)\ \Pi_2)$
$mt_{TPC}\ (rr, S)\ \Pi \ominus \mathcal{P} = (mt_{TPC}\ (rr, S)\ \Pi) - \mathcal{P}$
$mt_{TPC}\ (rr, S)\ \rho =$
    if $\rho \in rr$ then $S(\rho)$
    else let $\mathcal{P} = \bigcup_{\Pi_i \sqsubseteq \rho \in TPC} mt_{TPC}\ (rr \cup \{\rho\}, S)\ \Pi_i$
        in $\mathcal{P} \cup S(\rho)$

  $MT\ f\ \mathcal{P} = \mathcal{P}$
  $MT\ f\ \Pi_1 \sqcup \Pi_2 = f(\Pi_1) \cup f(\Pi_2)$
  $MT\ f\ \Pi \ominus \mathcal{P} = f(\Pi) - \mathcal{P}$
  $MT\ f\ \rho = \bigcup_{\Pi_i \sqsubseteq \rho \in TPC} f(\Pi_i)$
```

Figure 8: The mt_{TPC} algorithm and MT operator

THEOREM 4.8. If PC and TC are generated by *isecty*, then *satisfy*(PC, TC) holds iff PC, TC is satisfiable.

We determine whether PC, TC is satisfiable by providing the least solution to TC and checking it against the constraints in PC. The least solution of TC is the fixpoint of an appropriately defined operator MT, defined in figure 8. However, it is not immediately obvious that this operator has a fixed point at ω, so we define a function *glb* in figure 7 that clearly terminates, and show that it is the least fixpoint of MT.

In theorem 4.8 the stipulation that PC and TC are returned by *isecty* is important; the form of the constraints in such sets shapes our solution method. The following lemmas demonstrate that all set constraints imposed by TC are of the form $\Pi \sqsubseteq \rho$. Thus, the least solution of these constraints define a system of lower bounds on the size of each ρ.

LEMMA 4.9. If TC is returned by *isecty*, then for all $\tau <: \tau_1 \xrightarrow{\Pi} \tau_2 \in close(TC)$, Π is some set variable ρ and τ_2 is some type variable t.

PROOF. By induction on the number of closure rounds necessary to close TC. □

LEMMA 4.10. If $cTC = close(TC)$ for some TC returned by *isecty*, then for all $\Pi_1 \sqsubseteq \Pi_2 \in scs(cTC)$, Π_2 is a variable ρ.

PROOF. Immediate by lemma 4.9 and the definition of *scs*. □

Now, we note further that any constraint in PC will be of the form $\Pi \sqsubseteq \mathcal{P}$, so that to satisfy PC we want to find a "smallest possible" substitution; see lemma 4.18. Since the previous lemma establishes that TC imposes a system of lower bounds on set variables, the problem of satisfying PC, TC reduces to one of finding the glb of set variables ρ given TC, and then checking these bounds against the constraints imposed by PC. The function *glb* defined in figure 7 solves this problem, so the heart of the correctness proof of *isecty* examines its definition. More precisely, we analyze *glb* by way of the more general function mt_{TPC} defined in figure 8, which has features more amenable to our proof.

The essential steps in our proof use the functional operator MT given in figure 8; MT is defined in an environment with TPC bound to $scs(close(TC))$ where TC is returned by *isecty*. In particular, we demonstrate that any fixpoint of MT is a solution to the problem of finding a satisfying substitution for TPC (lemma 4.11), and then show that mt_{TPC} is a least fixpoint of MT (lemma 4.15). Our technique is inspired by the notion of an inductive definitions as described in [10], but is quite simple and requires none of the deeper theoretical results from that paper.

LEMMA 4.11. If f is a fixpoint of MT, there exists a substition S that satisfies TPC such that $f = \lfloor S \rfloor$.

PROOF. Let f be such that $MT(f) = f$; then it is easy to show that the first three clauses of MT ensure that there exists S such that $\lfloor S \rfloor = f$. And, by the fourth clause of MT it is the case that $[\![S(\Pi_i)]\!] \subseteq [\![S(\rho)]\!]$ for all ρ and $\Pi_i \sqsubseteq \rho \in TPC$, so that S satisfies TPC by lemma 4.9 and definition 3.4. □

Now we need to show that mt_{TPC} is a fixpoint of MT. The biggest issue to be dealt with here regards cycles in TPC. The following lemma establishes that if a recursive occurence of any ρ is encountered, we may substitute any value $\mathcal{P}$ for ρ at this point in computation, and the top-level evaluation grows monotonically in $\mathcal{P}$.

LEMMA 4.12. Let $mt_{TPC}\ (rr, S \circ \{\mathcal{P}/\rho\})\ \Pi = \mathcal{P}_1$ and $mt_{TPC}\ (rr, S \circ \{\mathcal{P} \cup \mathcal{P}'/\rho\})\ \Pi = \mathcal{P}_2$; then $\mathcal{P}_1 \subseteq \mathcal{P}_2 \subseteq \mathcal{P}_1 \cup \mathcal{P}'$.

PROOF. By induction on the call tree of $mt_{TPC}\ (rr, S \circ \{\mathcal{P}/\rho\})\ \Pi$. □

COROLLARY 4.13. If $S_1 \leq S_2$ and $\lfloor S_1' \rfloor = mt_{TPC}(\emptyset, S_1)$ and $\lfloor S_2' \rfloor = mt_{TPC}(\emptyset, S_2)$, then $S_1' \leq S_2'$. □

At this point we demonstrate the following lemma, which is essential for proving that mt_{TPC} is a fixpoint of MT:

LEMMA 4.14. Let $rr' = rr - \{\rho\}$, $\mathcal{P}_1 = mt_{TPC}\ (rr, S)\ \Pi$ and $\mathcal{P}_2 = mt_{TPC}\ (rr', S)\ \Pi$, and let $\mathcal{P} = mt_{TPC}\ (rr', S)\ \rho$; then $\mathcal{P}_1 \subseteq \mathcal{P}_2 \subseteq \mathcal{P}_1 \cup \mathcal{P}$.

PROOF. By induction on the call tree of $mt_{TPC}\ (rr, S)\ \Pi$. □

Having established the previous results, it is easy to prove that mt_{TPC} is a fixpoint of MT:

LEMMA 4.15. Let $S_\emptyset = \{\emptyset/\rho \mid \rho \in Vars(TPC \cup PC)\}$. Then $mt_{TPC}(\emptyset, S_\emptyset)$ is a fixpoint of MT.

PROOF. The proof proceeds by case analysis of arbitrary Π. In case that $\Pi = \mathcal{P}$, $\Pi_1 \sqcup \Pi_2$ or $\Pi' \ominus \mathcal{P}$, the lemma follows immediately by the definitions of mt_{TPC} and MT. Suppose on the other hand that $\Pi = \rho$; then by the definition of MT:

$$MT\ \ mt_{TPC}(\emptyset, S_\emptyset)\ \ \rho = \bigcup_{\Pi_i \sqsubseteq \rho \in TPC} mt_{TPC}\ \ (\emptyset, S_\emptyset)\ \ \Pi_i$$

and since $S_\emptyset(\rho) = \emptyset$, therefore by the definition of mt_{TPC}:

$$mt_{TPC}\ \ (\emptyset, S_\emptyset)\ \ \rho = \bigcup_{\Pi_i \sqsubseteq \rho \in TPC} mt_{TPC}\ \ (\{\rho\}, S_\emptyset)\ \ \Pi_i$$

Now, for each Π_i let $\mathcal{P}_i = mt_{TPC}\ (\emptyset, S_\emptyset)\ \Pi_i$, let $\mathcal{P}'_i = mt_{TPC}$ $(\{\rho\}, S_\emptyset)\ \Pi_i$, and let $\mathcal{P} = \cup_i \mathcal{P}_i$ and $\mathcal{P}' = \cup_i \mathcal{P}'_i$. By lemma 4.14, $\mathcal{P}'_i \subseteq \mathcal{P}_i \subseteq \mathcal{P}'_i \cup \mathcal{P}'$ for each i, so that $\mathcal{P}' \subseteq \mathcal{P} \subseteq \mathcal{P}'$, i.e. $\mathcal{P}' = \mathcal{P}$. Therefore, the lemma follows. □

The following lemma implies that mt_{TPC} is a least fixpoint of MT, as explicated in lemma 4.17:

LEMMA 4.16. If S satisfies TPC, then $mt_{TPC}(\emptyset, S) = \lfloor S \rfloor$.

PROOF. The proof proceeds by the claim that for all Π and rr it is the case that $mt_{TPC}\ (rr, S)\ \Pi = [\![S(\Pi)]\!]$, which follows by induction on the call tree of $mt_{TPC}\ (rr, S)\ \Pi$. Here we demonstrate only the case $(\Pi = \rho \in rr)$. In this case, $mt_{TPC}\ (rr, S)\ \Pi = (\bigcup_i mt_{TPC}\ (rr \cup \{\rho\}, S)\ \mathcal{P}_i) \cup S(\rho)$ over all $\Pi_i \sqsubseteq \rho \in TPC$. But by the induction hypothesis, $mt_{TPC}\ (rr \cup \{\rho\}, S)\ \mathcal{P}_i = [\![S(\Pi_i)]\!]$; and since S satisfies TPC by assumption, therefore $[\![S(\Pi_i)]\!] \subseteq [\![S(\rho)]\!]$ for each Π_i, so the lemma holds. □

Finally, we can show that mt_{TPC}, given the appropriate arguments, generates the glb of the privstructs in TPC:

LEMMA 4.17. There exists S such that $\lfloor S \rfloor = mt_{TPC}(\emptyset, S_\emptyset)$, and also S is the least substitution that satisfies TPC.

PROOF. By lemmas 4.11 and 4.15, there exists S such that $\lfloor S \rfloor = mt_{TPC}(\emptyset, S_\emptyset)$ which satisfies TPC. Furthermore, suppose some S' satisfies TPC. By lemma 4.16, $\lfloor S' \rfloor = mt_{TPC}(\emptyset, S')$, and by corrolary 4.13, $S \leq S'$, since clearly $S_\emptyset \leq S'$. Therefore, the lemma holds. □

The following lemma implies that if PC, TC is satisfiable, then the glb of TC must satisfy PC, TC:

LEMMA 4.18. If S satisfies PC and $S' \leq S$, then S' satisfies PC.

PROOF. By the definition of *isecty*, each constraint in PC is of the form $\Pi \sqsubseteq \mathcal{P}$. Thus, if S satisfies PC, then $[\![S(\Pi)]\!] \subseteq \mathcal{P}$ for each constraint in PC, so clearly if $S' \leq S$ then $[\![S'(\Pi)]\!] \subseteq \mathcal{P}$ for each constraint, by transitivity of $\subseteq$. □

Now we piece the preceding lemmas together to establish the correctness result for *satisfy*, and then the correctness result for type inference:

PROOF OF THEOREM 4.8 (CORRECTNESS OF *satisfy*) By definition of the algorithms, $glb(\Pi, \emptyset) = mt_{TPC}\ (\emptyset, S_\emptyset)\ \Pi$ with *glb* executing in an environment containing TPC. Thus, the result follows by the definition of *satisfy* and lemmas 4.17 and 4.18. □

PROOF OF THEOREM 4.4 (CORRECTNESS OF *infer_secty*) Immediate by the definition of *infer_secty*, definition 3.4, lemmas 4.6 and 4.7 and theorem 4.8. □

5. RELATED WORK

The use of static type systems to enforce or check security properties is a research area that has recently blossomed. None of the existing work applies to the context of fine-grained access control, as this paper does, but it shows that type-based techniques are emerging as a new methodology for securing information systems. Type systems for analyzing the security level of data via information flow have been developed [13, 11, 8], which are based on the earlier static analysis of [4]. This work is using a fundamentally different security model, the *information flow model*. The static information flow model of data security is derived from the classic Bell-LaPadula security model [2] which is a dynamic information flow model that ignores covert channels. Our proposed type system is analogous in that it is a static version of a dynamic model.

Another approach, related to the type approach, is proof-carrying code (PCC) [12]. In this paradigm, code is passed on the network along with a formal proof of a property of the code; the proof can then be mechanically checked at the server to verify that the property holds of the code. Type systems in this *certified code* framework can be viewed as *compressed proofs*—from the types it is possible to construct a typing proof, and program properties are implied by the proof. Walker in [14] has developed another sort of secure certified code, by describing "security automata" which can specify and enforce highly expressive security policies. However, certified code is in fact more general than our type-based approach—note that we focus on properties that can be inferred, whereas certified code approaches are often concerned with more complex ones.

Our type system design was inspired by Crary, Walker and Morrisett's static type system for memory region inference [3]. They share the commonality of addressing stack-based properties which are traditionally computed at run-time. Their paper does not use constraints or inference and instead defines an explicitly-typed system, an approach we also expect would work in this setting.

6. CONCLUSION AND FUTURE WORK

In this paper we have defined a type system which statically enforces security in a model of the Java JDK 1.2 security architecture. In the JDK, these properties are enforced by a dynamic stack inspection process. But the dynamic nature of stack inspection has several drawbacks, including the presence of a new class of run-time errors, a performance penalty, and lack of code readability due to the non-declarative nature of dynamic checks in code. A static, type-based approach eliminates these problems. Furthermore, the outermost types of functions or methods reflect their top-level needs, the privileges they need to execute. This means that security types are helpful in declaring the security policies of programs, since security requirements can be placed directly in the type signature.

We have proven a type safety result for our system, demonstrating that security types guarantee soundness with respect to stack inspection: run-time stack inspection will never fail on well-typed programs. Additionally, we have defined a type inference algorithm, meaning that the programmer need not explicitly declare security types. Our type inference algorithm is similar in function and complexity to standard constraint type inference algorithms. We have also proven our type inference algorithm correct, showing that it returns a most general type for any expression.

In this paper we have focused directly on the Java stack inspection model, as a well-understood point of departure. We have shown that a realistic model can be captured in a statically-typed framework. As future work, we intend to explore extensions to our system, including a security model which may improve on the JDK 1.2 model. In addition to simply adding more sophisticated features to the type system (e.g. polymorphism as discussed in subsection 3.5), we hope to develop a realistic language and type system to express, and enforce, the most effective security policies.

Acknowledgements

We would like to acknowledge Karl Crary for helpful discussions and François Pottier, Ran Rinat, and the ICFP referees for helpful comments on drafts of this paper.

7. REFERENCES

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference.z In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[2] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corp., Bedford MA, May 1973.

[3] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of the Twenty-sixth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, 1999.

[4] D. Denning. A lattice model of secure information flow. In *Communications of the ACM*, pages 236–243. ACM, May 1976.

[5] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. http://www.elsevier.nl/locate/entcs/volume1.html.

[6] L. Gong. Java Security Architecture (JDK1.2) . http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html, 1998.

[7] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, Dec. 1997.

[8] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 365–377, New York, N.Y., Jan. 1998. ACM.

[9] F. Henglein. Syntactic properties of polymorphic subtyping. TOPPS Technical Report (D-report series) D-293, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, May 1996.

[10] A. S. Kechris and Y. N. Moschovakis. Recursion in higher types. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundation of Mathematics*, chapter C.6, pages 681–737. North-Holland Publishing Company, 1977.

[11] X. Leroy and F. Rouaix. Security properties of typed applets. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium, San Diego, California, 19–21 January 1998*, pages 391–403, New York, NY 10036, USA, 1998. ACM Press.

[12] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, Oct. 1996. USENIX.

[13] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, Dec. 1996.

[14] D. Walker. A type system for expressive security policies. In *Twenty-seventh Symposium on Principles of Programming Languages*, pages 254–267, Boston, MA, January 2000. ACM SIGPLAN.

[15] D. S. Wallach. *A new Approach to Mobile Code Security*. PhD thesis, Princeton University, 1999.