A Systematic Approach to Static Access Control

François Pottier[1], Christian Skalka[2], and Scott Smith[2]

[1] INRIA Rocquencourt, Francois.Pottier@inria.fr
[2] The Johns Hopkins University, {ces,scott}@cs.jhu.edu

Abstract. The Java JDK 1.2 Security Architecture includes a dynamic mechanism for enforcing access control checks, so-called *stack inspection*. This paper studies type systems which can statically guarantee the success of these checks. We develop these systems using a new, systematic methodology: we show that the security-passing style translation, proposed by Wallach and Felten as a *dynamic* implementation technique, also gives rise to *static* security-aware type systems, by composition with conventional type systems. To define the latter, we use the general HM(X) framework, and easily construct several constraint- and unification-based type systems. They offer significant improvements on a previous type system for JDK access control, both in terms of expressiveness and in terms of readability of inferred type specifications.

1 Introduction

The Java Security Architecture [2], found in JDK 1.2 and later, includes mechanisms to protect systems from operations performed by untrusted code. These access control decisions are enforced by *dynamic* checks. Our goal is to make some or all of these decisions *statically*, by extensions to the type system. Thus, access control violations will be caught at compile-time rather than run-time. Furthermore, types (whether inferred or programmer-supplied) will constitute a specification of the security policy.

A Brief Review of the JDK Security Architecture For lack of space, we cover the JDK security architecture in a cursory manner here; see [2, 13, 8] for more detailed background. To use the access control system, the programmer adds `doPrivileged` and `checkPrivilege` commands to the code. At run-time, a `doPrivileged` command adds a flag to the current stack frame, enabling a particular privileged operation. The flag is implicitly eliminated when the frame is popped. When a privilege is checked via a `checkPrivilege` command, the stack frames are searched most to least recent. If a frame is encountered with the desired flag, the search stops and the check succeeds. Additionally, each stack frame is annotated with its owner (the owner of the method being invoked), and all stack frames searched by the above algorithm must be owned by some principal authorized for the privilege being checked. This keeps illicit code, invoked by the trusted codebase when `doPrivileged` is on the stack, from performing the privileged operation.

Our Framework This paper follows up on an initial access control type system presented by the last two authors in [8] and places emphasis on a more modular approach to type system construction. The previous paper developed the security type system *ab initio*. In this paper, we reduce the security typing problem to a conventional typing problem using a translation-based method inspired by [5]. We use a standard language of row types [7] to describe sets of privileges. We also re-use the HM(X) framework [3, 9], which allows a wide variety of type systems to be defined in a single stroke, saves some proof effort, and (most importantly) shows that our custom type systems arise naturally out of a standard one.

In addition to these methodological enhancements, this paper improves upon its predecessor in several other ways. In particular, [8] was based on subtyping constraints, whereas one of the type systems presented here uses row unification alone; this makes it more efficient and leads to more concise types. Also, the calculus studied in this paper allows for dynamic test-and-branch on whether a privilege is enabled. Lastly, because our new approach relies on HM(X), we can easily provide `let`-polymorphism.

We begin by defining a simplified model of the Java JDK 1.2 security architecture. It is a λ-calculus, called λ_{sec}, equipped with a non-standard operational semantics that includes a specification of stack inspection. In order to construct a static type system for λ_{sec}, we translate it into a standard λ-calculus, called λ_{set}. The translation is a security-passing style transformation [13]: it implements stack inspection by passing around sets of privileges at run-time. For this purpose, λ_{set} is equipped with built-in notions of set and set operations.

Then, we define a type system for λ_{set}. Because λ_{set} is a standard λ-calculus, we are able to define our type system as a simple instance of the HM(X) framework [3]. In fact, by using this framework a whole family of type systems may be succinctly defined, each with different costs and benefits. In order to give precise types to λ_{set}'s built-in set operations, our instance uses set types, defined as a simplification of Rémy's record types [7].

Lastly, we show that any type system for λ_{set} gives rise through the translation to a type system for λ_{sec}. The latter's correctness follows immediately from the former's, provided the translation itself is correct. This is quite easy to show, since the property does not involve types at all.

2 The Source Language $\boldsymbol{\lambda}_{\text{sec}}$

This section defines λ_{sec}, a simplified model of the JDK 1.2 security architecture. It is a λ-calculus equipped with a notion of code ownership and with constructs for enabling or checking privileges. Its grammar is given in Fig. 1.

We assume given notions of *principals* and *resources* (the latter also known as *privileges*), taken from arbitrary sets $\mathcal{P}$ and $\mathcal{R}$. We use p and r to range over principals and resources, respectively, and P and R to range over sets thereof.

We assume given a fixed *access credentials list* $\mathcal{A}$. It is a function which maps every principal $p \in \mathcal{P}$ to a subset of $\mathcal{R}$. We let $\mathcal{A}^{-1}$ denote its "inverse", that is,

$p \in \mathcal{P}, P \subseteq \mathcal{P}$	*principals*
$r \in \mathcal{R}, R \subseteq \mathcal{R}$	*resources*
$\mathcal{A} \in \mathcal{P} \to 2^{\mathcal{R}}$	*access credentials*
$v ::= \lambda x.f$	*values*
$e ::= x \mid \lambda x.f \mid e\,e \mid \mathrm{let}\,x = e\,\mathrm{in}\,e \mid \mathrm{letpriv}\,r\,\mathrm{in}\,e \mid$ $\mathrm{checkpriv}\,r\,\mathrm{for}\,e \mid \mathrm{testpriv}\,r\,\mathrm{then}\,e\,\mathrm{else}\,e \mid f$	*expressions*
$f ::= p.e$	*signed expressions*
$E ::= [] \mid E\,e \mid v\,E \mid \mathrm{let}\,x = E\,\mathrm{in}\,e \mid$ $\mathrm{letpriv}\,r\,\mathrm{in}\,E \mid p.E$	*evaluation contexts*

Fig. 1. Grammar for λ_{sec}

the function which maps a resource $r \in \mathcal{R}$ to $\{p \in \mathcal{P} \mid r \in \mathcal{A}(p)\}$. Without loss of generality, we assume the existence of a fixed principal p_0 such that $\mathcal{A}(p_0) = \varnothing$.

A signed expression $p.e$ behaves as the expression e endowed with the authority of principal p. Notice how the body of every λ-abstraction is required to be a signed expression – thus, every piece of code must be vouched for by some principal. The construct $\mathrm{letpriv}\,r\,\mathrm{in}\,e$ allows an authorized principal to enable the use of a resource r within the expression e. The construct $\mathrm{checkpriv}\,r\,\mathrm{for}\,e$ asserts that the use of r is currently enabled. If r is indeed enabled, e is evaluated; otherwise, execution fails. The construct $\mathrm{testpriv}\,r\,\mathrm{then}\,e_1\,\mathrm{else}\,e_2$ dynamically tests whether r is enabled, branching to e_1 or e_2 if this holds or fails, respectively.

2.1 Stack Inspection

The JDK 1.2 determines whether a resource is enabled by literally examining the runtime stack, hence the name *stack inspection*. We give a simple specification of this process by noticing that stacks are implicitly contained in *evaluation contexts*, whose grammar is defined in Fig. 1. Indeed, a context defines a path from the term's root down to its active redex, along which one finds exactly the security annotations which the JDK 1.2 would maintain on the stack, that is, code owners p and enabled resources r.

To formalize this idea, we associate a finite string of principals and resources, called a *stack*, to every evaluation context E. The right-most letters in the string correspond to the most recent stack frames.

$$\begin{aligned} \mathrm{stack}([]) &= \epsilon & \mathrm{stack}(E\,e) &= \mathrm{stack}(E) \\ \mathrm{stack}(v\,E) &= \mathrm{stack}(E) & \mathrm{stack}(\mathrm{let}\,x = E\,\mathrm{in}\,e) &= \mathrm{stack}(E) \\ \mathrm{stack}(\mathrm{letpriv}\,r\,\mathrm{in}\,E) &= r.\mathrm{stack}(E) & \mathrm{stack}(p.E) &= p.\mathrm{stack}(E) \end{aligned}$$

Then, Fig. 2 defines stack inspection, with $S \vdash r$ meaning access to resource r is allowed by stack S, and $S \vdash P$ meaning some principal in P is the most recent owner on S. This specification corresponds roughly to Wallach's [13, p. 71]. We write $E \vdash r$ for $\mathrm{stack}(E) \vdash r$.

$$\frac{r \in \mathcal{A}(p) \qquad S \vdash r}{S.p \vdash r} \qquad \frac{S \vdash r}{S.r' \vdash r} \qquad \frac{S \vdash \mathcal{A}^{-1}(r)}{S.r \vdash r} \qquad \frac{S \vdash P}{S.r \vdash P} \qquad \frac{p \in P}{S.p \vdash P}$$

Fig. 2. Stack inspection algorithm

2.2 Operational Semantics for $\boldsymbol{\lambda}_{\text{sec}}$

The operational semantics of λ_{sec} is defined by the following reduction rules:

$$\begin{array}{rcll}
E[(\lambda x.f)\, v] & \rightarrow & E[f[v/x]] & \\
E[\text{let}\, x = v \,\text{in}\, e] & \rightarrow & E[e[v/x]] & \\
E[\text{checkpriv}\, r \,\text{for}\, e] & \rightarrow & E[e] & \text{if } E \vdash r \\
E[\text{testpriv}\, r \,\text{then}\, e_1 \,\text{else}\, e_2] & \rightarrow & E[e_1] & \text{if } E \vdash r \\
E[\text{testpriv}\, r \,\text{then}\, e_1 \,\text{else}\, e_2] & \rightarrow & E[e_2] & \text{if } \neg(E \vdash r) \\
E[\text{letpriv}\, r \,\text{in}\, v] & \rightarrow & E[v] & \\
E[p.v] & \rightarrow & E[v] &
\end{array}$$

The first two rules are standard. The next rule allows $\text{checkpriv}\, r \,\text{for}\, e$ to reduce into e only if stack inspection succeeds (as expressed by the side condition $E \vdash r$); otherwise, execution is blocked. The following two rules use stack inspection in a similar way to determine how to reduce $\text{testpriv}\, r \,\text{then}\, e_1 \,\text{else}\, e_2$; however, they never cause execution to fail. The last two rules state that security annotations become unnecessary once the expression they enclose has been reduced to a value. In a Java virtual machine, these rules would be implemented simply by popping stack frames (and the security annotations they contain) after executing a method.

This operational semantics constitutes a concise, formal description of Java stack inspection in a higher-order setting. It is easy to check that every closed term either is a value, or is reducible, or is of the form $E[\text{checkpriv}\, r \,\text{for}\, e]$ where $\neg(E \vdash r)$. Terms of the third category are *stuck*; they represent access control violations. An expression e is said to *go wrong* if and only if $e \rightarrow^\star e'$, where e' is a stuck expression, holds.

3 The Target Calculus $\boldsymbol{\lambda}_{\text{set}}$

We now define a standard calculus, λ_{set}, to be used as the target of our translation. It is a λ-calculus equipped with a number of constants which provide set operations, and is given in Fig. 3. We will use $e.r$, $e \vee R$ and $e \wedge R$ as syntactic sugar for $(._r\, e)$, $(\vee_R\, e)$ and $(\wedge_R\, e)$, respectively.

The constant R represents a constant set. The construct $e.r$ asserts that r is an element of the set denoted by e; its execution fails if that is not the case. The construct $e \vee R$ (resp. $e \wedge R$) allows computing the union (resp. intersection) of the set denoted by e with a constant set R. Lastly, the expression $?_r\, e\, f\, g$

$$\begin{array}{rlr} e & ::= x \mid v \mid e\,e \mid \mathrm{let}\, x = e \,\mathrm{in}\, e & \textit{expressions} \\ v & ::= \lambda x.e \mid R \mid ._r \mid ?_r \mid \vee_R \mid \wedge_R & \textit{values} \\ E & ::= [] \mid E\,e \mid v\,E \mid \mathrm{let}\, x = E \,\mathrm{in}\, e & \textit{evaluation contexts} \end{array}$$

Fig. 3. Grammar for λ_{set}

dynamically tests whether r belongs to the set R denoted by e, and accordingly invokes f or g, passing R to it. The operational semantics for λ_{set} is as follows:

$$\begin{array}{rll} (\lambda x.e)\,v & \to e[v/x] & \\ \mathrm{let}\, x = v \,\mathrm{in}\, e & \to e[v/x] & \\ R.r & \to R & \text{if } r \in R \\ ?_r\, R & \to \lambda f.\lambda g.(f\,R) & \text{if } r \in R \\ ?_r\, R & \to \lambda f.\lambda g.(g\,R) & \text{if } r \notin R \\ R_1 \vee R_2 & \to R_1 \cup R_2 & \\ R_1 \wedge R_2 & \to R_1 \cap R_2 & \\ E[e] & \to E[e'] & \text{if } e \to e' \end{array}$$

Again, an expression e is said to *go wrong* if and only if $e \to^\star e'$, where e' is a stuck expression, holds.

4 Source-to-Target Translation

A translation of λ_{sec} into λ_{set} is defined in Fig. 4. The distinguished identifiers s and $\_$ are assumed not to appear in source expressions. Notice that s may appear free in translated expressions. Translating an (unsigned) expression requires specifying the current principal p.

One will often wish to translate an expression under minimal hypotheses, i.e. under the initial principal p_0 and a void security context. To do so, we define $(\!|\, e \,|\!) = [\![e]\!]_{p_0}[\varnothing/s]$. Notice that s does not appear free in $(\!|\, e \,|\!)$. If e is closed, then so is $(\!|\, e \,|\!)$.

$$\begin{array}{rl} [\![x]\!]_p & = x \\ [\![\lambda x.f]\!]_p & = \lambda x.\lambda s.[\![f]\!] \\ [\![e_1\, e_2]\!]_p & = [\![e_1]\!]_p\, [\![e_2]\!]_p\, s \\ [\![\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2]\!]_p & = \mathrm{let}\, x = [\![e_1]\!]_p \,\mathrm{in}\, [\![e_2]\!]_p \\ [\![\mathrm{letpriv}\, r \,\mathrm{in}\, e]\!]_p & = \mathrm{let}\, s = s \vee (\{r\} \cap \mathcal{A}(p)) \,\mathrm{in}\, [\![e]\!]_p \\ [\![\mathrm{checkpriv}\, r \,\mathrm{for}\, e]\!]_p & = \mathrm{let}\, \_ = s.r \,\mathrm{in}\, [\![e]\!]_p \\ [\![\mathrm{testpriv}\, r \,\mathrm{then}\, e_1 \,\mathrm{else}\, e_2]\!]_p & = ?_r\, s\, (\lambda s.[\![e_1]\!]_p)\, (\lambda s.[\![e_2]\!]_p) \\ [\![f]\!]_p & = [\![f]\!] \\ & \\ [\![p.e]\!] & = \mathrm{let}\, s = s \wedge \mathcal{A}(p) \,\mathrm{in}\, [\![e]\!]_p \end{array}$$

Fig. 4. Source-to-Target Translation

The idea behind the translation is simple: the variable s is bound at all times to the set of currently enabled resources. Every function accepts s as an extra parameter, because it must execute within its caller's security context. As a result, every function call has s as its second parameter. The constructs $\mathrm{letpriv}\, r \,\mathrm{in}\, e$ and $p.e$ cause s to be locally bound to a new value, reflecting the new security context; more specifically, the former enables r, while the latter disables all privileges not in $\mathcal{A}(p)$. The constructs $\mathrm{checkpriv}\, r \,\mathrm{for}\, e$ and $\mathrm{testpriv}\, r \,\mathrm{then}\, e_1 \,\mathrm{else}\, e_2$ are implemented simply by looking up the current value of s. In the latter, s is re-bound, within each branch, to the *same* value. This may appear superfluous at first sight, but has an important impact on typing, because it allows s to be given a different (more precise) type within each branch.

This translation can be viewed as a generalization of Wallach's security-passing style transformation [13] to a higher-order setting. Whereas they advocated this idea as an implementation technique, with efficiency in mind, we use it only as a vehicle in the proof of our type systems. Here, efficiency is not at stake. Our objective is only to define a correct translation, that is, to prove the following:

Theorem 4.1. *If $e \rightarrow^\star v$, then $(\!| e |\!) \rightarrow^\star (\!| v |\!)$. If e goes wrong, then $(\!| e |\!)$ goes wrong. If e diverges, then $(\!| e |\!)$ diverges.*

The proof is divided in two steps. First, we define a new stack inspection algorithm, which walks the stack forward instead of backward, and computes, at each step, the set of currently enabled resources. Then, we show that the translation implements this algorithm, interleaved with the actual code. Both proof steps are straightforward, and we omit them here for brevity.

5 Types for $\boldsymbol{\lambda}_{\mathrm{set}}$

We define a type system for the target calculus as an instance of the parametric framework HM(X) [3, 9]. HM(X) is a generic type system in the Hindley-Milner tradition, parameterized by an abstract constraint system X. Sect. 5.1 briefly recalls its definition. Sect. 5.2 defines a specific constraint system called SETS, yielding the type system HM(SETS). Sect. 5.3 extends HM(SETS) to the entire language λ_{set} by assigning types to its primitive operations. Sect. 5.4 states type safety results and discusses a couple of choices.

5.1 The System HM(X)

The system HM(X) is parameterized by a *sound term constraint system* X, i.e. by notions of *types* τ, *constraints* C, and *constraint entailment* $\Vdash$, which must satisfy a number of axioms [3].

Then, a *type scheme* is a triple of a set of quantifiers $\bar{\alpha}$, a constraint C, and a type τ (which, in this paper, must be of kind *Type*; see Sect. 5.2), written $\sigma ::= \forall\bar{\alpha}[C].\tau$. A *type environment* Γ is a partial mapping of program variables to type schemes. A *judgement* is a quadruple of a satisfiable constraint C, a

$$\frac{\Gamma(x) = \forall\bar{\alpha}[D].\tau \qquad C \Vdash \exists\bar{\alpha}.D}{C, \Gamma \vdash x : \forall\bar{\alpha}[D].\tau}\ \textsc{Var} \qquad \frac{C, \Gamma \vdash e : \tau \qquad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash e : \tau'}\ \textsc{Sub}$$

$$\frac{C, (\Gamma; x : \tau) \vdash e : \tau'}{C, \Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}\ \textsc{Abs} \qquad \frac{C, \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \qquad C, \Gamma \vdash e_2 : \tau_2}{C, \Gamma \vdash e_1\, e_2 : \tau}\ \textsc{App}$$

$$\frac{C, \Gamma \vdash e_1 : \sigma \qquad C, (\Gamma; x : \sigma) \vdash e_2 : \tau}{C, \Gamma \vdash \mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2 : \tau}\ \textsc{Let} \qquad \frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathrm{fv}(C, \Gamma) = \varnothing}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau}\ \forall\ \textsc{Intro}$$

$$\frac{C, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau}{C \wedge D, \Gamma \vdash e : \tau}\ \forall\ \textsc{Elim} \qquad \frac{C, \Gamma \vdash e : \sigma \qquad \bar{\alpha} \cap \mathrm{fv}(\Gamma, \sigma) = \varnothing}{\exists\bar{\alpha}.C, \Gamma \vdash e : \sigma}\ \exists\ \textsc{Intro}$$

Fig. 5. The system HM(X)

$$\begin{array}{rll}
\tau ::= & \alpha, \beta, \ldots \mid \tau \rightarrow \tau \mid \{\tau\} \mid r : \tau ;\ \tau \mid \partial\tau \mid c & \textit{types} \\
c ::= & \mathbf{NA} \mid \mathbf{Pre} \mid \mathbf{Abs} \mid \mathbf{Either} & \textit{capabilities} \\
C ::= & \mathbf{true} \mid C \wedge C \mid \exists\alpha.C \mid \tau = \tau \mid \tau \leq \tau \mid \text{if } c \leq \tau \text{ then } \tau \leq \tau & \textit{constraints}
\end{array}$$

Fig. 6. SETS Grammar

type environment Γ, an expression e and a type scheme σ, written $C, \Gamma \vdash e : \sigma$, derivable using the rules of Fig. 5. These rules correspond to those given in [9].

The following type safety theorem is proven in [3] with respect to a denotational presentation of the call-by-value λ-calculus with `let`. We have proved a syntactic version of it, in the style of [14], which better suits our needs.

Theorem 5.1. *If $C, \Gamma \vdash e : \sigma$ holds, then e does not go wrong.*

5.2 The Constraint System SETS

In order to give precise types to the primitive set operations in λ_{set}, we need specific types and constraints. Together with their logical interpretation, which defines their meaning, these form a constraint system called SETS.

The syntax of types and constraints is defined in Fig. 6. The type language features a *set* type constructor $\{\cdot\}$, the two standard *row* constructors [7], and four *capability* constructors. Capabilities tell whether a given element may appear in a set (**Pre**), may not appear in it (**Abs**), may or may not appear in it (**Either**), or whether this information is irrelevant, because the set itself is unavailable (**NA**). For instance, the singleton set $\{r\}$ will be one (and the only)

$$\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \qquad \frac{\tau, \tau' : \mathit{Type}}{\tau \to \tau' : \mathit{Type}} \qquad \frac{\tau : \mathit{Row}_\varnothing}{\{\tau\} : \mathit{Type}} \qquad \frac{\tau : \mathit{Cap} \quad r \notin R \quad \tau' : \mathit{Row}_{R \cup \{r\}}}{(r : \tau \,;\, \tau') : \mathit{Row}_R} \qquad \frac{\tau : \mathit{Cap}}{\partial\tau : \mathit{Row}_R}$$

$$\frac{}{c : \mathit{Cap}} \qquad \frac{}{\vdash \mathbf{true}} \qquad \frac{\vdash C_1, C_2}{\vdash C_1 \wedge C_2 \quad \vdash \text{if } C_1 \text{ then } C_2} \qquad \frac{\vdash C}{\vdash \exists \alpha.C} \qquad \frac{\tau, \tau' : k}{\vdash \tau = \tau' \quad \vdash \tau \leq \tau'}$$

Fig. 7. Kinding rules

value of type $\{r : \mathbf{Pre}\,;\, \partial\mathbf{Abs}\}$. The constraint language offers standard equality and subtyping constraints, as well as a form of conditional constraints. Sample uses of these types and constraints will be shown in Sect. 5.3.

To ensure that only meaningful types and constraints can be built, we immediately equip them with *kinds*, defined by $k ::= \mathit{Cap} \mid \mathit{Row}_R \mid \mathit{Type}$, where R ranges over finite subsets of $\mathcal{R}$. For every kind k, we assume given a distinct, denumerable set of *type variables* $\mathcal{V}_k$. We use $\alpha, \beta, \gamma, \ldots$ to represent type variables. From here on, we consider only *well-kinded* types and constraints, as defined in Fig. 7. The purpose of these rules is to guarantee that every constraint has a well-defined interpretation within our model, whose definition follows.

To every kind k, we associate a mathematical structure $[\![k]\!]$. $[\![\mathit{Cap}]\!]$ is the set of all four capabilities. Given a finite set of resources $R \subseteq \mathcal{R}$, $[\![\mathit{Row}_R]\!]$ is the set of total, almost constant functions from $\mathcal{R} \setminus R$ into $[\![\mathit{Cap}]\!]$. (A function is *almost constant* if it is constant except on a finite number of inputs.) In short, Row_R is the kind of rows which do *not* carry the fields mentioned in R; $\mathit{Row}_\varnothing$ is the kind of complete rows. $[\![\mathit{Type}]\!]$ is the free algebra generated by the constructors $\to$, with signature $[\![\mathit{Type}]\!] \times [\![\mathit{Type}]\!] \to [\![\mathit{Type}]\!]$, and $\{\cdot\}$, with signature $[\![\mathit{Row}_\varnothing]\!] \to [\![\mathit{Type}]\!]$.

Each of these structures is then equipped with an ordering. Here, a choice has to be made. If we do *not* wish to allow subtyping, we merely define the ordering on every $[\![k]\!]$ as equality. Otherwise, we proceed as follows. First, a lattice over $[\![\mathit{Cap}]\!]$ is defined, whose least (resp. greatest) element is **NA** (resp. **Either**), and where **Abs** and **Pre** are incomparable. This ordering is then extended, pointwise and covariantly, to every $[\![\mathit{Row}_R]\!]$. Finally, it is extended inductively to $[\![\mathit{Type}]\!]$ by viewing the constructor $\{\cdot\}$ as covariant, and the constructor $\to$ as contravariant (resp. covariant) in its first (resp. second) argument.

We may now give the interpretation of types and constraints within the model. It is parameterized by an *assignment* ρ, i.e. a function which, for every kind k, maps $\mathcal{V}_k$ into $[\![k]\!]$. The interpretation of types is obtained by extending ρ so as to map every type of kind k to an element of $[\![k]\!]$, as follows:

$$\begin{array}{rlr} \rho(\tau \to \tau') = \rho(\tau) \to \rho(\tau') & \rho(\{\tau\}) = \{\rho(\tau)\} & \\ \rho(r : \tau\,;\, \tau')(r) = \rho(\tau) & \rho(r : \tau\,;\, \tau')(r') = \rho(\tau')(r') & (r \neq r') \\ \rho(\partial\tau)(r) = \rho(\tau) & \rho(c) = c & \end{array}$$

$$\frac{}{\rho \vdash \mathbf{true}} \qquad \frac{\rho \vdash C_1 \qquad \rho \vdash C_2}{\rho \vdash C_1 \wedge C_2} \qquad \frac{\rho = \rho' \,[\alpha] \qquad \rho' \vdash C}{\rho \vdash \exists \alpha . C}$$

$$\frac{\rho(\tau) = \rho(\tau')}{\rho \vdash \tau = \tau'} \qquad \frac{\rho(\tau) \leq \rho(\tau')}{\rho \vdash \tau \leq \tau'} \qquad \frac{c \leq \rho(\tau) \Rightarrow \rho \vdash \tau' \leq \tau''}{\rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}$$

Fig. 8. Interpretation of constraints

Fig. 8 defines the constraint satisfaction predicate $\cdot \vdash \cdot$, whose arguments are an assignment ρ and a constraint C. (The notation $\rho = \rho' \,[\alpha]$ means that ρ and ρ' coincide except possibly on α.) *Entailment* is defined as usual: $C \Vdash C'$ (read: C entails C') holds iff, for every assignment ρ, $\rho \vdash C$ implies $\rho \vdash C'$.

We refer to the type and constraint logic, together with its interpretation, as SETS. More precisely, we have defined two logics, where $\leq$ is interpreted as either equality or as a non-trivial subtype ordering. We will refer to them as $\text{SETS}^{=}$ and $\text{SETS}^{\leq}$, respectively. Both are sound term constraint systems [3].

5.3 Dealing with the Primitive Operations in λ_{set}

The typing rules of HM(X) cover only the λ-calculus with `let`. To extend HM(SETS) to the whole language λ_{set}, we must assign types to its primitive operations. Let us define an initial type environment Γ_1 as follows:

$$\begin{aligned}
R &: \{R : \mathbf{Pre}\,;\ \partial\mathbf{Abs}\} \\
\cdot_r &: \forall\beta.\{r : \mathbf{Pre}\,;\ \beta\} \rightarrow \{r : \mathbf{Pre}\,;\ \beta\} \\
\vee_R &: \forall\beta\bar{\gamma}.\{R : \bar{\gamma}\,;\ \beta\} \rightarrow \{R : \mathbf{Pre}\,;\ \beta\} \\
\wedge_R &: \forall\beta\bar{\gamma}.\{R : \bar{\gamma}\,;\ \beta\} \rightarrow \{R : \bar{\gamma}\,;\ \partial\mathbf{Abs}\} \\
?_r &: \forall\alpha\beta\gamma.\{r : \gamma\,;\ \beta\} \rightarrow (\{r : \mathbf{Pre}\,;\ \beta\} \rightarrow \alpha) \rightarrow (\{r : \mathbf{Abs}\,;\ \beta\} \rightarrow \alpha) \rightarrow \alpha
\end{aligned}$$

Here, α, β, γ range over type variables of kind *Type*, $\mathit{Row}_\star$, *Cap*, respectively. We abuse notation: if R is $\{r_1, \ldots, r_n\}$, then $R : c$ denotes $r_1 : c\,;\ \ldots\,;\ r_n : c$, and $R : \bar{\gamma}$ denotes $r_1 : \gamma_1\,;\ \ldots\,;\ r_n : \gamma_n$.

None of the type schemes in Γ_1 carry constraints. If we wish to take advantage of conditional constraints, we must refine the type of $?_r$. Let Γ_2 be the initial type environment obtained by replacing the last binding in Γ_1 with

$$\begin{aligned}
?_r : \forall\bar{\alpha}\bar{\beta}\gamma[C].\{r : \gamma\,;\ \beta\} &\rightarrow (\{r : \mathbf{Pre}\,;\ \beta_1\} \rightarrow \alpha_1) \rightarrow (\{r : \mathbf{Abs}\,;\ \beta_2\} \rightarrow \alpha_2) \rightarrow \alpha \\
\text{where } C = \quad & \text{if } \mathbf{Pre} \leq \gamma \text{ then } \beta \leq \beta_1 \wedge \text{ if } \mathbf{Abs} \leq \gamma \text{ then } \beta \leq \beta_2 \\
\wedge \ & \text{if } \mathbf{Pre} \leq \gamma \text{ then } \alpha_1 \leq \alpha \wedge \text{ if } \mathbf{Abs} \leq \gamma \text{ then } \alpha_2 \leq \alpha
\end{aligned}$$

Here, the input and output of each branch (represented by β_i and α_i, respectively) are linked to the input and output of the whole construct (represented by β and α) through conditional constraints. Intuitively, this means that the security requirements and the return type of a branch may be entirely ignored unless the branch seems liable to be taken. (For more background on conditional constraints, the reader is referred to [1, 4].)

5.4 The Type Systems $\mathcal{S}_i^{rel}$

Sect. 5.2 describes two constraint systems, SETS$^=$ and SETS$^\leq$. Sect. 5.3 defines two initial typing environments, Γ_1 and Γ_2. These choices give rise to four related type systems, which we refer to as $\mathcal{S}_i^{rel}$, where *rel* and i range over $\{=, \leq\}$ and $\{1, 2\}$, respectively. Each of them offers a different compromise between accuracy, readability and cost of analysis. In each case, Theorem 5.1 may be extended to the entire language λ_{set} by proving a simple *δ-typability* [14] lemma, i.e. by checking that Γ_i correctly describes the behavior of the primitive operations. The proofs are straightforward and are not given here.

Despite sharing a common formalism, these systems may call for vastly different implementations. Indeed, every instance of HM(X) must come with a constraint resolution algorithm. $\mathcal{S}_1^=$ is a simple extension of the Hindley-Milner type system with rows, and may be implemented using unification [6]. $\mathcal{S}_2^=$ is similar, but requires conditional (i.e. delayed) unification constraints, adding some complexity to the implementation. $\mathcal{S}_1^\leq$ and $\mathcal{S}_2^\leq$ require maintaining subtyping constraints, usually leading to complex implementations.

In the following, we lack the space to describe all four variants. Therefore, we will focus on $\mathcal{S}_1^=$. Because it is based on unification, it is efficient, easy to implement, and yields readable types. We conjecture that, thanks to the power of row polymorphism, it is flexible enough for many practical uses (see Sect. 7.3).

6 Types for λ_{sec}

6.1 Definition

Sect. 5 defined a type system, $\mathcal{S}_i^{rel}$, for λ_{set}. Sect. 4 defined a translation of λ_{sec} into λ_{set}. Composing the two automatically gives rise to a type system for λ_{sec}, also called $\mathcal{S}_i^{rel}$ for simplicity, whose safety is a direct consequence of Theorems 4.1 and 5.1.

Definition 6.1. *Let e be a closed λ_{sec} expression. By definition, $C, \Gamma \vdash e : \sigma$ holds if and only if $C, \Gamma \vdash (\!| e |\!) : \sigma$ holds.*

Theorem 6.2. *If $C, \Gamma \vdash e : \sigma$ holds, then e does not go wrong.*

Turning type safety into a trivial corollary was the main motivation for basing our approach on a translation. Indeed, because Theorem 4.1 concerns untyped terms, its proof is straightforward. (The δ-typability lemmas mentioned in Sect. 5.3 do involve types, but are typically very simple.) A direct type safety proof would be non-trivial and would duplicate most of the steps involved in proving HM(X) correct.

6.2 Reformulation: Derived Type Systems

Definition 6.1, although simple, is not a direct definition of typing for λ_{sec}. We thus will give rules which allow typing λ_{sec} expressions without explicitly translating them into λ_{set}. These so-called *derived* rules can be obtained in a rather systematic way from the definition of $\mathcal{S}_i^{rel}$ and the definition of the translation. (In fact, it would be interesting to formally automate the process.)

In these rules, the symbols τ and ς range over types of kind *Type*; more specifically, ς is used to represent some security context, i.e. a set of available resources. The symbols ρ and φ range over types of kind $Row_\star$ and *Cap*, respectively. The $\star$ symbol in the rules indicates an irrelevant principal. In the source-to-target translation, all functions are given an additional parameter, yielding types of the form $\tau_1 \to \varsigma_2 \to \tau_2$. To recover the more familiar and appealing notation proposed in [8], we define the macro $\tau_1 \xrightarrow{\varsigma_2} \tau_2 =_{def} \tau_1 \to \varsigma_2 \to \tau_2$.

Fig. 9 gives derived rules for $\mathcal{S}_1^{=}$, the simplest of our type systems. There, all

$$\textsc{Var}\ \frac{\Gamma(x) = \sigma}{p, \varsigma, \Gamma \vdash x : \sigma} \qquad \textsc{Abs}\ \frac{\star, \varsigma_2, (\Gamma; x : \tau_1) \vdash f : \tau_2}{p, \varsigma_1, \Gamma \vdash \lambda x.f : \tau_1 \xrightarrow{\varsigma_2} \tau_2}$$

$$\textsc{App}\ \frac{p, \varsigma, \Gamma \vdash e_1 : \tau_2 \xrightarrow{\varsigma} \tau \qquad p, \varsigma, \Gamma \vdash e_2 : \tau_2}{p, \varsigma, \Gamma \vdash e_1\, e_2 : \tau}$$

$$\textsc{Let}\ \frac{p, \varsigma, \Gamma \vdash e_1 : \sigma \qquad p, \varsigma, (\Gamma; x : \sigma) \vdash e_2 : \tau}{p, \varsigma, \Gamma \vdash \text{let}\, x = e_1 \,\text{in}\, e_2 : \tau} \qquad \forall\ \textsc{Intro}\ \frac{p, \varsigma, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \text{fv}(\varsigma, \Gamma) = \varnothing}{p, \varsigma, \Gamma \vdash e : \forall \bar{\alpha}.\tau}$$

$$\forall\ \textsc{Elim}\ \frac{p, \varsigma, \Gamma \vdash e : \forall \bar{\alpha}.\tau}{p, \varsigma, \Gamma \vdash e : \tau[\bar{\tau}/\bar{\alpha}]} \qquad \textsc{Letpriv}^{-}\ \frac{p, \{\rho\}, \Gamma \vdash e : \tau \qquad r \notin \mathcal{A}(p)}{p, \{\rho\}, \Gamma \vdash \text{letpriv}\, r \,\text{in}\, e : \tau}$$

$$\textsc{Letpriv}^{+}\ \frac{p, \{r : \mathbf{Pre}\,;\ \rho\}, \Gamma \vdash e : \tau \qquad r \in \mathcal{A}(p)}{p, \{r : \varphi\,;\ \rho\}, \Gamma \vdash \text{letpriv}\, r \,\text{in}\, e : \tau} \qquad \textsc{Checkpriv}\ \frac{p, \{r : \mathbf{Pre}\,;\ \rho\}, \Gamma \vdash e : \tau}{p, \{r : \mathbf{Pre}\,;\ \rho\}, \Gamma \vdash \text{checkpriv}\, r \,\text{for}\, e : \tau}$$

$$\textsc{Testpriv}\ \frac{p, \{r : \mathbf{Pre}\,;\ \rho\}, \Gamma \vdash e_1 : \tau \qquad p, \{r : \mathbf{Abs}\,;\ \rho\}, \Gamma \vdash e_2 : \tau}{p, \{r : \varphi\,;\ \rho\}, \Gamma \vdash \text{testpriv}\, r \,\text{then}\, e_1 \,\text{else}\, e_2 : \tau}$$

$$\textsc{Own}\ \frac{p, \{r_1 : \varphi_1\,;\ \ldots\,;\ r_n : \varphi_n\,;\ \partial\mathbf{Abs}\}, \Gamma \vdash e : \tau \qquad \mathcal{A}(p) = \{r_1, \ldots, r_n\}}{\star, \{r_1 : \varphi_1\,;\ \ldots\,;\ r_n : \varphi_n\,;\ \rho\}, \Gamma \vdash p.e : \tau}$$

Fig. 9. Typing rules for λ_{sec} derived from $\mathcal{S}_1^{=}$

constraints are equations. As a result, all type information can be represented in term form, rather than in constraint form [9]. We exploit this fact to give a simple presentation of the derived rules. Type schemes have the form $\forall\bar{\alpha}.\tau$, and judgements have the form $p, \varsigma, \Gamma \vdash e : \sigma$.

To check that these derived rules are correct, we prove the following lemmas:

Lemma 6.3. *$p, \varsigma, \Gamma \vdash e : \sigma$ holds iff $\mathbf{true}, (\Gamma_1; \Gamma; s : \varsigma) \vdash [\![e]\!]_p : \sigma$ holds.*

Lemma 6.4. *$p_0, \{\partial\mathbf{Abs}\}, \Gamma \vdash e : \sigma$ holds iff $\mathbf{true}, (\Gamma_1; \Gamma) \vdash (\!| e |\!) : \sigma$ holds.*

Together, Theorem 6.2 and Lemma 6.4 show that, if a closed λ_{sec} expression e is well-typed according to the rules of Fig. 9, under the initial principal p_0 and the empty security context $\{\partial\mathbf{Abs}\}$, then e cannot go wrong.

Derived rules for each member of the $\mathcal{S}_i^{rel}$ family can be given in a similar way. The same process can also be used to yield type inference rules, rather than the logical typing rules shown here.

7 Examples

7.1 Basic Use of Security Checks

Imagine an operating system with two kinds of processes, root processes and user processes. Killing a user process is always allowed, while killing a root process requires the privilege *killing*. At least one distinguished principal *root* has this privilege. The system functions which perform the killing are implemented by *root*, as follows:

$$kill = \lambda(p : process).root.\text{checkpriv } killing \text{ for } \ldots () \quad - \textit{kill the process}$$
$$killIfUser = \lambda(p : process).root.\ldots () \quad - \textit{kill the process if it is user-level}$$

In system $\mathcal{S}_1^{=}$, these functions receive the following (most general) types:

$$kill : \forall\beta.process \xrightarrow{\{killing:\mathbf{Pre}\,;\,\beta\}} unit$$
$$killIfUser : \forall\beta.process \xrightarrow{\{\beta\}} unit$$

The first function can be called only if it can be statically proven that the privilege *killing* is enabled. The second one, on the other hand, can be called at any time, but will never kill a root process. To complement these functions, it may be desirable to define a function which provides a "best attempt" given the current (dynamic) security context. This may be done by dynamically checking whether the privilege is enabled, then calling the appropriate function:

$$tryKill = \lambda(p : process).root.$$
$$\text{testpriv } killing \text{ then } kill(p) \text{ else } killIfUser(p)$$

This function is well-typed in system $\mathcal{S}_1^{=}$. Indeed, within the first branch of the testpriv construct, it is statically known that the privilege *killing* must be enabled; this is why the sub-expression *kill*(p) is well-typed. The inferred type shows that *tryKill* does not have any security requirements:

$$tryKill : \forall\beta.process \xrightarrow{\{\beta\}} unit$$

7.2 Security Wrappers

A library writer often needs to surround numerous internal functions with "boilerplate" security code before making them accessible. To avoid redundancy, it seems desirable to allow the definition of generic *security wrappers*. When applied to a function, a wrapper returns a new function which has the same computational meaning but different security requirements.

Assume given a principal p such that $\mathcal{A}(p) = \{r, s\}$. Here are two wrappers likely to be of use to this principal:

$$\begin{aligned} \mathit{enable}_r &= \lambda f.p.\lambda x.p.\mathrm{letpriv}\, r \,\mathrm{in}\, f\, x \\ \mathit{require}_r &= \lambda f.p.\lambda x.p.\mathrm{checkpriv}\, r \,\mathrm{for}\, f\, x \end{aligned}$$

In system $\mathcal{S}_1^{=}$, these wrappers receive the following (most general) types:

$$\begin{aligned} \mathit{enable}_r &: \forall \ldots .(\alpha_1 \xrightarrow{\{r:\mathbf{Pre}\,;\, s:\gamma_1\,;\, \partial\mathbf{Abs}\}} \alpha_2) \xrightarrow{\{\beta_1\}} (\alpha_1 \xrightarrow{\{r:\gamma_2\,;\, s:\gamma_1\,;\, \beta_2\}} \alpha_2) \\ \mathit{require}_r &: \forall \ldots .(\alpha_1 \xrightarrow{\{r:\mathbf{Pre}\,;\, s:\gamma_1\,;\, \partial\mathbf{Abs}\}} \alpha_2) \xrightarrow{\{\beta_1\}} (\alpha_1 \xrightarrow{\{r:\mathbf{Pre}\,;\, s:\gamma_1\,;\, \beta_2\}} \alpha_2) \end{aligned}$$

These types are very similar; they may be read as follows. Both wrappers expect a function f which allows that r be enabled ($r : \mathbf{Pre}$), i.e. one which *either* requires r to be enabled, *or* doesn't care about its status. (Indeed, as in ML, the type of the actual argument may be more general than that of the formal.) They return a new function with identical domain and codomain (α_1, α_2), which works regardless of r's status (enable_r yields $r : \gamma_2$) or requires r to be enabled ($\mathit{require}_r$ yields $r : \mathbf{Pre}$). The new function retains f's expectations about s ($s : \gamma_1$). f must not require any further privileges ($\partial\mathbf{Abs}$), because it is invoked by p, which enjoys privileges r and s only.

These polymorphic types are very expressive. Our main concern is that, even though the privilege s is not mentioned in the *code* of these wrappers, it does appear in their *type*. More generally, every privilege in $\mathcal{A}(p)$ may show up in the type of a function written on behalf of principal p, which may lead to very verbose types. An appropriate type abbreviation mechanism may be able to address this problem; this is left as a subject for future work.

7.3 Advanced Examples

We lack space to cover numerous more subtle features of the type systems; let us give only some brief comments.

In Sect. 7.1, our use of testpriv was easily seen to be correct, because the sensitive action $\mathit{kill}(p)$ was performed within its lexical scope. Matters become more delicate when testpriv is used to yield a function (or, in Java, an object), whose security requirements *depend* on the test's outcome, and which is later invoked outside its scope. Conditional constraints are then required to track the dependency and prove that the function invocation is safe. It is not clear whether this idiom is a critical one to support in practice, and the question may be answerable only through experiment.

In Sect. 7.2, we pointed out that it is legal to pass $enable_r$ a function f which doesn't care about the status of r, provided the type of f is *polymorphic* in r's status, as in

$$\forall\gamma.\alpha_1 \xrightarrow{\{r:\gamma\,;\,\beta\}} \alpha_2$$

If, on the other hand, it is monomorphic (because f is λ-bound rather than `let`-bound), as in

$$\alpha_1 \xrightarrow{\{r:\mathbf{Either}\,;\,\beta\}} \alpha_2$$

then the application ($enable_r\ f$) becomes well-typed only if subtyping is available, i.e. if **Pre** is a subtype of **Either**. We expect this situation to be infrequent, although this remains to be confirmed.

8 Discussion

Extension to a Full-Featured Language Many features of the Java language or environment are not addressed in this theoretical study. In particular, Java views privileges as first-class objects, making static typing problematic. In our model, privileges are identifiers, and expressions cannot compute privileges. In the case of Java, it is an open question whether a completely static mechanism can be devised. If not, it may be desirable to take a soft typing approach [1].

Related Work The security-passing style translation described in Sect. 4 is monadic. Monadic type systems have been used to analyze the use of impure language features in otherwise pure languages [11]. However, as deplored in [11], there is still "a need to create a new effect system for each new effect". In other words, we apparently cannot readily re-use the work on monadic type systems in our setting. In fact, our work may be viewed as a *systematic* construction of an "effect" type system adapted to our particular effectful programming language.

Several researchers have proposed ways of defining efficient, provably correct compilation schemes for languages whose security policy is expressed by a *security automaton*. Walker [12] defines a source language, equipped with such a security policy, then shows how to compile it into a dependently-typed target language, whose type system, by encoding assertions about security states, guarantees that no run-time violations will occur. Walker first builds the target type system, then defines a typed translation. On the opposite, our approach consists in first defining an untyped translation, then letting the source type system arise from it. Thiemann's approach to security automata [10] is conceptually much closer to ours: he also starts with an untyped security-passing translation, whose output he then feeds through a standard program specializer, in order to automatically obtain an optimizing translation.

Our paper shares some motivations with these works; however, our aim was not only to gain performance by eliminating many dynamic checks, but also to define a programming discipline. This requires security types to be available not only at the level of compiled code, as in Walker's work, but also in the source code itself.

References

[1] Alexander S. Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Principles of Programming Languages*, pages 163–173, January 1994. URL: `http://http.cs.berkeley.edu/~aiken/ftp/popl94.ps`.

[2] Li Gong. Java security architecture (JDK1.2). URL: `http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html`, October 1998.

[3] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. URL: `http://www.cs.mu.oz.au/~sulzmann/publications/tapos.ps`.

[4] François Pottier. A 3-part type inference engine. In Gert Smolka, editor, *Proceedings of the 2000 European Symposium on Programming (ESOP'00)*, volume 1782 of *Lecture Notes in Computer Science*, pages 320–335. Springer Verlag, March 2000. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-esop-2000.ps.gz`.

[5] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, September 2000. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-conchon-icfp00.ps.gz`.

[6] Didier Rémy. Extending ML type system with a sorted equational theory. Technical Report 1766, INRIA, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, 1992. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/eq-theory-on-types.ps.gz`.

[7] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM Press. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/lfp92.ps.gz`.

[8] Christian Skalka and Scott Smith. Static enforcement of security with types. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 34–45, Montréal, Canada, September 2000. URL: `http://www.cs.jhu.edu/~ces/papers/secty_icfp2000.ps.gz`.

[9] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999. URL: `http://www.ps.uni-sb.de/~mmueller/papers/hm-constraints.ps.gz`.

[10] Peter Thiemann. Enforcing security properties using type specialization. In David Sands, editor, *Proceedings of the 2001 European Symposium on Programming (ESOP'01)*, Lecture Notes in Computer Science. Springer Verlag, April 2001.

[11] Philip Wadler and Peter Thiemann. The marriage of effects and monads. Submitted to *ACM Transactions on Computational Logic*. URL: `http://cm.bell-labs.com/cm/cs/who/wadler/papers/effectstocl/effectstocl.ps.gz`.

[12] David Walker. A type system for expressive security policies. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267, Boston, Massachusetts, January 2000. URL: `http://www.cs.cornell.edu/home/walker/papers/sa-popl00_ps.gz`.

[13] Dan S. Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, January 1999. URL: `http://www.cs.princeton.edu/sip/pub/dwallach-dissertation.html`.

[14] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994. URL: `http://www.cs.rice.edu/CS/PLT/Publications/ic94-wf.ps.gz`.