A Type System for Expressive Security Policies*

David Walker
Cornell University

Abstract

Certified code is a general mechanism for enforcing security properties. In this paradigm, untrusted mobile code carries annotations that allow a host to verify its trustworthiness. Before running the agent, the host checks the annotations and proves that they imply the host's security policy. Despite the flexibility of this scheme, so far, compilers that generate certified code have focused on simple type safety properties rather than more general security properties.

Security automata can specify an expressive collection of security policies including access control and resource bounds. In this paper, we describe how to instrument well-typed programs with security checks and typing annotations. The resulting programs obey the policies specified by security automata and can be mechanically checked for safety. This work provides a foundation for the process of automatically generating certified code for expressive security policies.

1 Introduction

Strong type systems such as those of Java or ML provide provable guarantees about the run-time behaviour of programs. If we type check programs before executing them, we know they "won't go wrong." Usually, the notion "won't go wrong" implies *memory safety* (programs only access memory that has been allocated for them), *control flow safety* (programs only jump to and execute valid code), and *abstraction preservation* (programs use abstract data types only as their interfaces allow). These properties are essential building blocks for any secure system such as a web browser, extensible operating system, or server that may download, check and execute untrusted programs. However, standard type safety properties alone do not enforce access control or restrict the dissemination of secret information.

*This material is based on work supported in part by the AFOSR grant F49620-97-1-0013 and in part by the National Science Foundation under Grant No. EIA 97-03470. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

Certified code is a general framework for verifying security properties in untrusted programs. To use this security architecture, programmers and compilers must attach a collection of annotations to the code they produce. These annotations can be proofs, types, or encodings of some other kind of formal system. Regardless, there must be some way of reconstructing a proof that the code obeys a certain security policy, for upon receiving annotated code, an untrusting web browser or operating system will use a mechanical checker to verify that the code is safe before executing it. For example, Necula and Lee's proof-carrying code (PCC) implementation [20, 19] uses a first-order logic and they have shown that they can check many interesting properties of hand-coded assembly language programs including access control and resource bound policies [22].

In order to make certified code a practical technology, it must be possible to generate code and certificate automatically. However, so far, *certifying compilers* have focused on a limited set of properties. For example, the main focus of Necula and Lee's proof-generating compiler Touchstone [21] is the creation of efficient code; the security policy it enforces is the standard type and memory safety. Other frameworks, including Kozen's efficient code certification (ECC) [8] and Morrisett *et al.*'s Typed Assembly Language (TAL) [18, 17], concentrate exclusively on standard type safety properties.

The main reason that certified code has been used in this restricted fashion is that automated theorem provers are not powerful enough to infer properties of arbitrary programs and constructing proofs by hand is prohibitively expensive. Unable to prove security properties statically, real-world security systems such as the Java Virtual Machine (JVM) [12] have fallen back on run-time checking. Dynamic security checks are scattered throughout the Java libraries and are intended to ensure that applets do not access protected resources inappropriately. However, this situation is unsatisfying for a number of reasons:

- The security checks and therefore the security policy is distributed throughout library code instead of being specified declaratively in a centralized location. This can make the policy difficult to understand or change.
- There is no way to verify that checks have not been forgotten or misplaced.
- There is no way to optimize away redundant checks and certify that the resulting code is safe.

Figure 1 presents a practical alternative for producing certified code. First, we give a formal specification of a security policy. Then, during compilation, the security policy dictates when to insert run-time checks. For example, the security policy might state that programs cannot use the network after they have read any local files. In this case, the compiler places run-time checks around every call to a network operation to ensure the program does not use the network after reading a file. Next, the program is optimized, removing run-time checks whereever it is safe to do so. However, each time the optimizer removes a check from the program, it must leave behind enough information to be able to verify that calling the function is still safe.

When compilation has been completed, software will verify that the resulting code is safe and then link the application program with the rest of the system. In order to ensure the security policy is obeyed, the software will obviously have to use the formal policy specification. It can either do this directly or, as shown in Figure 1, it can compile the high-level interface into a low-level interface and use the specification to annotate the result. For example, if the high-level system interface contains a network API, and the security policy is the one described above, an automated tool might compile the API into some low-level format and annotate the low-level code with pre-conditions requiring that the application has not yet read a file. When the verifier checks an application that uses the network, it must be able to prove these pre-conditions are satisfied.

This framework has a number of advantages over other security architectures. First, by defining the security policy independently of the rest of the system, it may be simpler for the implementer of the security system to write or change the policy, and easier for the applications programmers to understand it. The fact that the security policy can be defined in terms of a high-level system interface rather than its low-level implementation will also make the policy easier to understand.

Second, for a large class of security properties, a compiler can use run-time checks to ensure that any program can be *automatically* rewritten so that it is provably secure. This mechanism relieves the programmer of the burden of constructing proofs that his or her programs obey the security policy. In fact, although applications programmers need to be aware of and to follow the security policy (otherwise their programs will be terminated prematurely), the high-level programming model need not change at all.

Third, there is clear separation between the trusted and untrusted components of the system. More precisely, we must trust that:

- The formal security policy specification implies the desired semantic properties.
- The annotated low-level system interface enforces the formal security policy specification and the code that performs run-time checks is correct.
- The verification software is correct.

However, because verification is independent of the instrumentation algorithm, we do not have to trust the application program or its compiler. Untrusted parties can write their own application-specific instrumentation and optimization transformations. In fact, programmers do not even have to use the same source language. They can write programs in different languages and then compile them into a common intermediate form or even write low-level code by hand. Regardless of the source of the application code, it can and will be verified before it is linked into the extensible system.

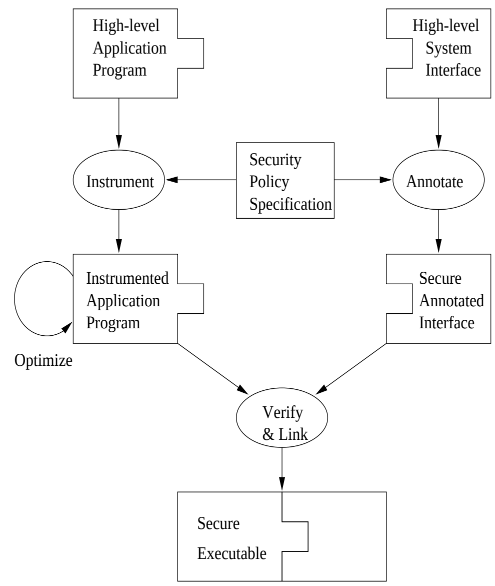

Figure 1: Architecture of a Certifying Compiler

2 Overview

In this paper, we explore one particular instance of the general framework:

- Our source language is *type-safe*. Type safety enforces many essential safety properties statically without having to resort to run-time checking.
- Our formal security policies are specified using *security automata* [24, 27]. We have chosen security automata because they are very expressive, able to encode any *safety property* [24]. Moreover, security automaton specifications can always be enforced by inserting run-time checks. Given a security automaton specification, a compiler can automatically rewrite any source language program so that it obeys the security policy.
- Our target language is dependently-typed. We encode the security automaton definition in the types of the security-sensitive operations. The type system is powerful enough to enforce any security automaton specification and yet is flexible enough to allow a number of optimizations.

Why use a type system? There are other frameworks for static verification. For example, Necula and Lee use a verification-condition generator that emits proof obligations in the form of first-order logic formulae with some extensions. Such a framework has clear advantages: First-order logic is extremely expressive and yet has simple proof rules. One advantage of type theory is that it handles higher-order features such as first-class functions and continuations naturally. This fact is particularly important when handling low-level languages because even the results of compiling procedural languages without function pointers are naturally higher-order: When a piece of machine code "calls" or jumps to a procedure, it passes the return address explicitly to the code it jumps to and therefore every machine-language code fragment can be thought of as a higher-order function. Higher-order type constructors are also very useful in low-level langauges. They can help compress redundant information and are used extensively in TAL to speed type checking [15].

In practice, there are ways to work around some of these difficulties and stay within a first-order system. For example, Necula and Lee fix the calling convention in their PCC implementation so they do not have to treat return addresses as first-class functions. However, this decision prevents them from performing some optimizations such as tail-call elimination. In order to keep proof sizes small, Necula and Lee have extended conventional first-order logic with high-level language-specific axioms.

Perhaps the most compelling reason for using a type system is to be able to draw upon the vast literature on type-directed compilation. This large body of knowledge certainly lead researchers to focus on certifying compilers for type-safety properties in the first place. We already know how to preserve many kinds of typing invariants from top to bottom, through the compilation of (higher-order) modules [5, 10], closure conversion [13] and code generation [18]. It is less clear how to preserve proofs represented in other formalisms through these transformations. Moreover, there has been much research on implementing efficient, decidable type checking and type reconstruction algorithms for practical programming languages. By choosing a type system as the foundation for security, we can immediately take advantage of existing these implementation techniques and theoretical results. Of course, using a type system throughout compilation does not prevent implementers from encoding the final low-level results in another logic or logical framework, if they choose to do so.

The remaining sections of this paper describe our approach to security in more detail. First, in Section 3, we present security automata, our mechanism for specifying security policies. Our presentation is derived from work by Úlfar Erlingsson and Fred Schneider [24, 27]. Next, Section 4 defines a generic typed target language for our compiler. This language is a dependently-typed lambda calculus that can encode any security automaton specification. After defining the generic language, we explain how to automatically construct a typed system interface that will specialize the language so that it enforces the policy of a particular automaton. Section 5 shows how to automatically instrument programs written in a high-level language (the simply-typed lambda calculus) and explores optimizations to the procedure. The last section discusses additional related work.

3 Security Automata

In our system, security policies are specified using *security automata* [24]. These automata are defined similarly to other automata [7]: They possess a set of states and rules for making transitions from one state to the next. The author of the security policy determines the set of states and the transition relation. One of these states is designated as the *bad* state and entry into this state is a violation of the security policy.

Security automata enforce safety properties by monitoring programs as they execute. Before an untrusted program is allowed to execute a security-sensitive or *protected* operation, the security automaton checks to see if the operation will cause a transition to the *bad* state. If so, the automaton terminates the program. If not, the program is allowed to execute the operation and the security automaton makes a transition to a new state. For example, a web browser might allow applets to open and read files, and send bits on the network. However, assuming the web browser wants to ensure some level of privacy, the open, read, and send operations will be designated protected operations. Before an untrusted applet can execute one of these functions, the browser will check the security automaton definition to ensure the operation is allowed in the current state.

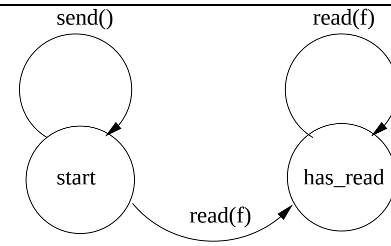

Figure 2: Security Automaton for a File System Policy

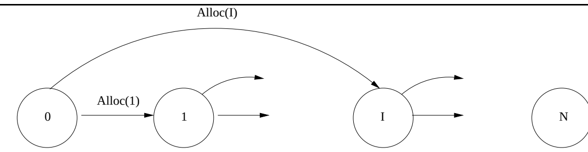

Figure 3: Security Automaton for a Memory Bounds Policy

By monitoring programs in this way, security automata are sufficiently powerful that they can restrict access to private files or bound the use of resources. They can also enforce the safety properties typically implied by type systems such as memory safety and control-flow safety. In fact, security automata can enforce any *safety property* [24]. However, some interesting security policies including information flow, resource availability, and general liveness properties are not safety properties and cannot be enforced by this mechanism. Nevertheless, Schneider [24] points out that some of these applications can still use a security automaton: The automaton must simply enforce a stronger property than is required. For example, the policy that admits any program that allocates a finite amount of memory cannot be enforced by a security automaton. However, security automata can enforce the policy that prevents a program from allocating more than an *a priori* fixed amount (such as 1 MB) of memory. The latter policy reduces the number of legal programs that can be written (a program that allocates 1.1 MB will be prematurely terminated), but it may be effective in practice.

Figure 2 depicts a security automaton that enforces the simple policy that programs must not perform a send on the network after reading a file. The security automaton actually has three states: the *start* state, the *has_read* state, and the *bad* state, which is not shown in the diagram. For the purpose of this example, there are only two protected operations: the *send* operation and the *read* operation. Each of the arcs in the graph is labeled with one of these operations and indicates the state transition that occurs when the operation is invoked. If there is no outgoing arc from a certain state labeled with the appropriate operation, the automaton makes a transition to the *bad* state. For example, there is no *send* arc emanating from the *has_read* state. Consequently, if a program tries to use the network in the *has_read* state, it will be terminated.

Figure 3 shows a second security automaton that restricts the amount of heap memory that a program can al-

Q	a finite or countably infinite set of states
q_0	a distinguished initial state
bad	the single "bad" state
F	a finite set of function symbols (f)
A	a countable set of constants (a)
δ	a computable (deterministic) total function: $\text{F} \to (Q \times \vec{\text{A}}) \to Q$

Figure 4: Elements of a Security Automaton

locate. The states are indexed by the natural numbers 0 through N and the protected operation $alloc(I)$ causes an automaton transition from state S to state $S + I$, provided $S + I$ is less than or equal to N.

3.1 Formal Definitions

For our purposes, a security automaton ($\mathcal{A}$) is a 6-tuple containing the fields summarized in Figure 4. Like other automata, a security automaton has a set of states Q (either finite or a countably infinite), and a distinguished initial state q_0. The automaton also has a single bad state (bad). Entrance into the bad state indicates that the security policy has been violated. All other states are considered "good" or "accepting" states. The automaton's inputs correspond to the application of a function f to arguments $a_1, \ldots, a_n$ where f is taken from the set F (the set of protected operations) and $a_1, \ldots, a_n$ are taken from A.

A security automaton defines allowable behaviour by specifying a transition function δ. Formally, δ is a deterministic, total function with a signature $\text{F} \to (Q \times \vec{\text{A}}) \to Q$ where $\vec{\text{A}}$ denotes the set of lists $a_1, \ldots, a_n$. Upon receiving an input $f(a_1, \ldots, a_n)$, an automaton makes a transition from its current state to the next state as dictated by this transition function. If the security policy permits the operation f on arguments $a_1, \ldots, a_n$ in the current state q then the next state, $\delta(f)(q, a_1, \ldots, a_n)$, will be one of the "good" ones. On the other hand, if the security policy disallows the operation then $\delta(f)(q, a_1, \ldots, a_n)$ will equal bad. Furthermore, once the automaton enters the bad state, it stays there. Formally, for all f and $a_1, \ldots, a_n$, $\delta(f)(bad, a_1, \ldots, a_n) = bad$.

The transition function δ must also be computable. Moreover, the implementor of the security policy must supply code for a family of functions δ_f such that $\delta_f(q_1, a_1, \ldots, a_n)$ equals $\delta(f)(q_1, a_1, \ldots, a_n)$. In the following sections, we will use these functions to instrument untrusted code with security checks.

The language accepted by the automaton $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$, is a set of strings where a string is a finite sequence of symbols $s_1, \ldots, s_n$ and each symbol s_i is of the form $f(a_1, \ldots, a_m)$. The string $s_1, \ldots, s_n$ belongs to $\mathcal{L}(\mathcal{A})$ if the predicate $Accept(q_0, s_1, \ldots, s_n)$ holds where $Accept$ is the least predicate such that:

Definition 1 (String Acceptance) *For all states q and (possibly empty) sequences of symbols $s_1, \ldots, s_n$, $Accept(q, s_1, \ldots, s_n)$ if $q \neq bad$ and:*

1. *$s_1, \ldots, s_n$ is the empty sequence or*
2. *$s_1 = f(a_1, \ldots, a_m)$ and $\delta(f)(q, a_1, \ldots, a_m) = q'$ and $Accept(q', s_2, \ldots, s_n)$*

3.2 The File System Policy

We can now define the file system policy described in the previous section. Infomally, the policy states that no network send is allowed after any file is read. For the sake of future examples, we extend this policy slightly by restricting access to some files; in order to determine whether or not the applet has access to the file a, we will have to invoke the function $read?(a)$, which has been provided by the implementor of the file system.

The corresponding security automaton has three states: $start$, the initial state; has_read, the state we enter after any file read; and, of course, the bad state. The protected operations, F, are $send$ and $read$ and the constants, A, include all files. The transition function δ, written in pseudo-code, is the following:

$$
\begin{aligned}
\delta_{send}(q) \; = \;& \\
& \texttt{if}\; q = start \;\texttt{then} \\
& \quad start \\
& \texttt{else} \\
& \quad bad
\end{aligned}
$$

$$
\begin{aligned}
\delta_{read}(q, a) \; = \;& \\
& \texttt{if}\; q = start \wedge read?(a) \;\texttt{then} \\
& \quad has\_read \\
& \texttt{else if}\; q = has\_read \wedge read?(a) \;\texttt{then} \\
& \quad has\_read \\
& \texttt{else} \\
& \quad bad
\end{aligned}
$$

3.3 Enforcing Security Automaton Policies

We can enforce the policies specified by security automata by instrumenting programs with run-time security checks. For example, according to the file system policy from the previous section, we must be in the $start$ state in order for the $send$ to be safe. A program instrumentation tool could enforce this policy by wrapping the code invoking $send$ with security checks:

```
let next = δ_send(current) in
if next = bad then halt
else send()
```

The first statement invokes the security automaton to determine the next state given that a $send$ operation is invoked in the current state. The second statement tests the next state to make sure it is not the bad one. If it is bad, then program execution is terminated using the halt instruction.

After performing the initial transformation that wraps all protected operations with run-time security checks, a program optimizer might attempt to eliminate redundant checks by performing standard program optimizations such as loop-invariant removal and common subexpression elimination. An optimizer might also use its knowledge of the special structure of a security automaton to eliminate more checks than would otherwise be possible.

An implementation developed by Erlingsson and Schneider [27] attests to the fact that security automata can enforce a broad, practical set of security policies. Their tool, SASI, automatically instruments untrusted code with checks dictated by a security automaton specification and optimizes the output code to eliminate checks that can be proven unnecessary. SASI is both flexible and efficient and they have implemented a variety of security policies from the literature. For example, using SASI for the Intel Pentium architecture, they have specified the memory and control-flow safety policy enforced by Software Fault Isolation (SFI) [28]. The SASI-instrumented code is only slightly slower than the code produced by the special-purpose, hand-coded MiSFIT tool for SFI [25]. As another example, using SASI for the Java Virtual Machine, they have been able to reimplement the security manager for Sun's Java 1.1. The SASI-instrumented code is equally as efficient as the Java security manager in some cases and more efficient in others. Because the SASI security policy is specified separately from the rest of the Java system, it is simpler to modify than in the current system.

4 A Secure Typed Target Language

The primary goal of the type system is to ensure that the protected operations are used in accordance with the security policy. In order to accomplish this task, we have taken an approach similar to Dependent ML [33] and embedded a logic into the type system. The logic described in this section is intentionally kept simple for the sake of the presentation, but it is not necessary to restrict an implementation to this level of simplicity. Section 5.2 explains how to extend the logic with some security-policy specific axioms that allow several optimizations. Implementers should extend the logic further, if they find the need.

The type system and its logic allows a compiler to give each protected operation a type that specifies a pre-condition related to the security policy. This pre-condition must be proven before the function can be called. For example, in order to enforce our file system policy, the type of the *send* operation could include the following predicates:

$$P_1 : \delta_{send}(current, next)$$
$$P_2 : next \neq bad$$

They state that executing the *send* operation causes a transition from the state *current* to the state *next*, and moreover, that the *next* is not *bad*. Therefore, if the automaton is in the state *current*, and we can prove these predicates, code can call the *send* operation and the security policy will not be violated. Upon return, the type of the *send* operation should indicate that the automaton is in state *next*.

In general, given an arbitrary program, a type checker will not be able to prove all pre-conditions on protected operations. These pre-conditions depend upon the current state of the automaton and statically tracking the automaton state through arbitrary code is undecidable. However, whenever the type checker cannot prove a pre-condition, a compiler can insert a run-time check. If we can reflect the result of the dynamic check into the static type system, then proving the pre-conditions will be straight-forward.

The following example demonstrates how to verify that the wrapper code for the *send* operation is safe, assuming information from dynamic tests (such as *if*) can be reflected into the type system:

$\texttt{let}\ next\ \texttt{=}\ \delta_{send}(current)\ \texttt{in}$
$P_1\!: \quad \delta_{send}(current, next)$
$\texttt{if}\ next\ \texttt{=}\ bad\ \texttt{then}\ \cdots$
$\texttt{else}$
$\qquad P_2\!: next \neq bad$
$\qquad send()$

Predicate P_1 is the post-condition for the function δ_{send}, which implements the automaton transition function. The predicates P_1 and P_2, together with the information that the automaton is in the state *current*, are sufficient to prove the pre-condition on the *send* operation.

In order to ensure the last condition is met (*i.e.* that the automaton is actually in the state described by the data structure *current*), the type system cannot trust the programmer to manipulate the state data structures properly. For instance, the type system must prevent a malicious programmer from substituting some unrelated state $current'$ for the proper state *current* in the code above. The type system prevents such behaviour by statically keeping track of the current state itself. Below, the comments $state : X$ indicate what the type checker knows about the current state. Initially, it knows the current state is *current*. Only after calling *send* does it believe the state has changed to *next*:

$state : current$	(*)
$\texttt{let}\ next\ \texttt{=}\ \delta_{send}(current)\ \texttt{in}$	(**)
$state : current$	
$\texttt{if}\ next\ \texttt{=}\ bad\ \texttt{then halt}$	$state : current$
$\texttt{else}\ send()$	$state : next$

So far during this informal explanation, we have been a little sloppy. On line (**) above, *current* is a value variable that is used *at run-time*, but on line (*), *current* is part of a *compile-time* expression (and similarly for *current*, *next* and δ_{send} in the previous code fragment). However, it is essential to make a clear distinction between the compile-time portion of the language and the run-time portion of the language. In the code above, successful compile-time verification was only possible after the run-time check was inserted. Nevertheless, run-time checks are not always necessary, and therefore it is not always necessary to physically pass the automaton state around. On the other hand, a sound type system must *always* track the state statically. Therefore, if run-time objects are not distinguished from compile-time objects, code will always pass the automaton state around at run-time, even when it is only used for compile-time verification. Since we are concerned with optimization as well as safety, we would rather not force this property on the language.

A second important reason to make a clean separation between run-time and compile-time expressions is to ensure that the compile-time component is decidable. If the compile-time logic is a small subset of the run-time expressions, then it will be easy to decide. However, if run-time and compile-time are inextricably mixed together, as they are in some dependently-typed languages, then type checking will be as hard as deciding arbitrary program equivalence. Therefore, from now on, we will use hat notation to distinguish between compile-time ($\hat{x}$) and run-time objects (x) whenever there might be ambiguity.

Despite this separation, certain run-time values, such as the state data structure *current*, must be reflected into the static type system. The type system includes *singleton types* for this purpose. For example, if a variable x has the singleton type $\texttt{S}(\widehat{start})$, then the type system is not only aware that x represents a state, but also that x represents the particular state *start*.

Unfortunately, when used in isolation, singleton types are actually *too precise* for most situations. In the example above, the reason for the dynamic test was that the type system could not statically track the automaton state precisely enough! Perhaps this program point occurs just after a join in the code or a return from an unknown function. In these cases, it may be impossible to give the state a precise type like $\texttt{S}(\widehat{start})$.

kinds	κ	$::=$	$\texttt{Val} \mid \texttt{State} \mid \mathcal{B} \mid (\kappa, \ldots, \kappa) \to \mathcal{B}$
indices	$\iota, \hat{a}, \hat{q}$		
index sig	$\mathcal{I}$	$:$	*indices* $\to$ *kinds*
pred. vars	ρ, ϱ		
predicates	P	$::=$	$\rho \mid \iota \mid \iota(P_1, \ldots, P_n)$
predicate ctxt	Δ	$::=$	$\cdot \mid \Delta, \rho{:}\kappa \mid \Delta, P$
base types	b	$\in$	$\text{BaseType} + \texttt{S}$
types	τ	$::=$	$b(P) \mid \forall[\Delta].(P, \tau_1, \ldots, \tau_n) \to \texttt{0} \mid \exists\rho{:}\kappa.\tau$
constants	a, q, f		
constant sig	$\mathcal{C}$	$:$	*constants* $\to$ *types*
value vars	g, x		
values	v	$::=$	$x \mid a \mid \texttt{fix}\, g[\Delta].(P, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \mid v[P] \mid v[\cdot] \mid \texttt{pack}[P, v]\ \textit{as}\ \tau$
expressions	e	$::=$	$v_0(v_1, \ldots, v_n) \mid \texttt{halt} \mid \texttt{let}\, \rho, x = \texttt{unpack}\, v\, \texttt{in}\, e \mid \texttt{if}\, v\, (q \to e_1 \mid \_ \to e_2)$

Figure 5: Target Language Syntax

Our solution to this problem is standard: Add polymorphism. After a join in the code, the exact state may be unknown statically. However, the type system can describe that state using a type variable ϱ. If code has access to a physical object with type $\texttt{S}(\varrho)$, then it can test the physical object against a known state. For example, if x has type $\texttt{S}(\varrho)$, code can dynamically test x against the state *bad* using the expression $\texttt{if}\ x = \textit{bad}\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. In this case, the expression e_1 is checked in a context where *bad* has been substituted for the variable ϱ, reflecting equality information, and the expression e_2 is checked in a context including the predicate $\textit{next} \neq \textit{bad}$, reflecting disequality information. In both branches, the most precise information about the result of the run-time test is reflected into the type system.

Once we have added singleton types for automaton states, it is straight-forward to add singleton types for other values as well. These singletons will help encode policies that depend upon values other than the current state. Such policies are very common. For example, in our simple file system policy, the right to read a file depends upon the file that is requested for reading. In the memory bounds policy, the right to allocate memory depends upon both upon the amount of memory that has been allocated already (the current automaton state) and on how much more memory is being requested (the argument to the *alloc* function).

Finally, in order to allow values with singleton types to be stored in data structures, such as arrays or lists, without making the type of the array or list too precise, the type system includes existential types. More generally, escaping singletons with type $\texttt{S}(P)$ can be encapsulated using an existential type $\exists\varrho{:}\texttt{State}.\texttt{S}(\varrho)$, and used generically. For example, the type $\exists\varrho{:}\texttt{Val}.\textit{int}(\varrho)$ can be used as an unspecific, ordinary integer throughout the "security-insensitive" code (such as the math library, for instance), but when the appropriate time comes (the code invokes a security-sensitive operation), the existential can be opened up and the necessary dependencies created. We use this property to simplify the formal instrumentation procedure.

4.1 Formal Definitions

Formally, the target language contains three main parts: the predicates P, the types τ, and the term level constructs. We will explain each of these parts in succession. Figure 5 presents the syntax of the entire language.

Predicates Predicates, P, may be variables ρ or ϱ, indices (constant predicates) ι, or functions of a number of arguments $\iota(P_1, \ldots, P_n)$. All of these predicates are compile-time only objects. Predicates are further divided into three distinct kinds:

- Predicates that correspond to values of base type.[1] For each value a of base type, there is a corresponding index $\hat{a}$ of kind $\texttt{Val}$. By convention, we use ρ for variables of kind $\texttt{Val}$.
- Predicates that correspond to security automaton states. For each automaton state q, there is a corresponding index $\hat{q}$ of kind $\texttt{State}$. By convention, we use ϱ for variables of kind $\texttt{State}$.
- Predicates that describe relations between values and/or states. These predicates have kind $(\kappa_1, \ldots, \kappa_n) \to \mathcal{B}$ where $\mathcal{B}$ is the boolean kind.

The last kind of predicate serves two purposes. Predicates of the form $\delta_{send}(P_{q_1}, P_{q_2})$ describe the automaton transition function; they may be read "in state P_{q_1}, executing the send operation causes a transition to state P_{q_2}." A special predicate $\neq(\cdot,\cdot)$ of kind $(\texttt{State}, \texttt{State}) \to \mathcal{B}$ will be used to denote the fact that some state q is not equal to the *bad* state and consequently that it is safe to execute an operation that causes a transition into q.

We specify the well-formedness of predicates using the judgement $\Phi \vdash P : \kappa$ where Φ is a type-checking context containing three components: a predicate context Δ, a finite map Γ from value variables to types, and another predicate P' indicating the current state of the automaton. The latter two components are not used for specifying the well-formedness of predicates (they will be used for type-checking terms). The signature $\mathcal{I}$ assigns kinds to the indices. Kinds for variables are determined by the predicate context Δ. The kind of a function symbol must agree with the kinds of the predicates to which it is applied. Due to space considerations, the formal rules have been omitted (see the technical report [29] for details).

Figure 6 gives the rules for proving predicates. The judgement $\Phi \vdash P$ indicates that the boolean-valued predicate P is true, and the judgement $\Phi \vdash P\ \text{in\_state}$ indicates that the automaton is in state P.

[1] In order to simplify the presentation, we only consider policies that depend upon values of base type here. See Section 4.4 for further explanation.

$\boxed{\Phi \vdash P \qquad \Phi \vdash P \text{ in\_state}}$

$$\frac{}{\Delta_1, P, \Delta_2; \Gamma; P' \vdash P} \tag{1}$$

$$\frac{}{\Phi \vdash \neq (\hat{q}, \hat{q'})} \; (\hat{q} \neq \hat{q'}) \tag{2}$$

$$\frac{}{\Delta; \Gamma; P \vdash P \text{ in\_state}} \tag{3}$$

$\boxed{\Phi \vdash v : \tau}$

$$\frac{}{\Phi \vdash x : \tau} \; (\Phi(x) = \tau) \tag{4}$$

$$\frac{}{\Phi \vdash a : \tau} \; (\mathcal{C}(a) = \tau) \tag{5}$$

$$\frac{\begin{array}{c}\Phi \vdash \tau_g \\ (\Phi, \Delta, g{:}\tau_g, x_1{:}\tau_1, \ldots, x_n{:}\tau_n) \leftarrow P \vdash e\end{array}}{\Phi \vdash \mathtt{fix}\, g[\Delta].(P, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \tau_g} \tag{6}$$

(where $\tau_g = \forall[\Delta](P, \tau_1, \ldots, \tau_n) \to \mathtt{0}$)

$$\frac{\Phi \vdash v : \forall[\rho{:}\kappa, \Delta].(P', \tau_1, \ldots, \tau_n) \to \mathtt{0} \quad \Phi \vdash P : \kappa}{\Phi \vdash v[P] : (\forall[\Delta].(P', \tau_1, \ldots, \tau_n) \to \mathtt{0})[P/\rho]} \tag{7}$$

$$\frac{\Phi \vdash v : \forall[P, \Delta].(P', \tau_1, \ldots, \tau_n) \to \mathtt{0} \quad \Phi \vdash P}{\Phi \vdash v[\cdot] : \forall[\Delta].(P', \tau_1, \ldots, \tau_n) \to \mathtt{0}} \tag{8}$$

$$\frac{\Phi \vdash P : \kappa \quad \Phi \vdash v : \tau[P/\rho]}{\Phi \vdash \mathtt{pack}[P, v] \; as \; \exists\rho{:}\kappa.\tau : \exists\rho{:}\kappa.\tau} \tag{9}$$

$\boxed{\Phi \vdash e}$

$$\frac{\begin{array}{c}\Phi \vdash v_0 : \forall[].(P, \tau_1, \ldots, \tau_n) \to \mathtt{0} \\ \Phi \vdash v_1 : \tau_1 \quad \cdots \quad \Phi \vdash v_n : \tau_n \\ \Phi \vdash P \text{ in\_state}\end{array}}{\Phi \vdash v_0(v_1, \ldots, v_n)} \tag{10}$$

$$\frac{}{\Phi \vdash \mathtt{halt}} \tag{11}$$

$$\frac{\Phi \vdash v : \exists\rho{:}\kappa.\tau \quad \Phi, \rho{:}\kappa, x{:}\tau \vdash e}{\Phi \vdash \mathtt{let}\, \rho, x = \mathtt{unpack}\, v \,\mathtt{in}\, e} \tag{12}$$

$$\frac{\begin{array}{c}\Phi \vdash v : \mathtt{S}(\rho) \qquad \Phi \vdash q : \mathtt{S}(\hat{q}) \\ \Phi' \vdash e_1[\hat{q}/\rho] \qquad \Phi, \neq (\rho, \hat{q}) \vdash e_2\end{array}}{\Phi \vdash \mathtt{if}\, v\, (q \to e_1 \mid \_ \to e_2)} \tag{13}$$

(where $\Phi = \Delta, \rho{:}\mathtt{State}, \Delta'; \Gamma; P$ and $\Phi' = \Delta, (\Delta'[\hat{q}/\rho]); \Gamma[\hat{q}/\rho]; P[\hat{q}/\rho]$)

$$\frac{\begin{array}{c}\Phi \vdash v : \mathtt{S}(\hat{q}) \qquad \Phi \vdash q : \mathtt{S}(\hat{q}) \\ \Phi \vdash e_1\end{array}}{\Phi \vdash \mathtt{if}\, v\, (q \to e_1 \mid \_ \to e_2)} \tag{14}$$

$$\frac{\begin{array}{c}\Phi \vdash v : \mathtt{S}(P) \qquad \Phi \vdash q : \mathtt{S}(\hat{q}) \\ \Phi, \neq (P, \hat{q}) \vdash e_2\end{array}}{\Phi \vdash \mathtt{if}\, v\, (q \to e_1 \mid \_ \to e_2)} \; \begin{pmatrix} P \neq \rho \\ P \neq \hat{q} \end{pmatrix} \tag{15}$$

Figure 6: Target Language Static Semantics

Aside from the special predicate $\neq (\cdot, \cdot)$, our predicates are completely uninterpreted. This decision makes it trivial to show the decidability of the type system. However, some optimizations may not be possible without a stronger logic. To remedy this situation, implementers may add axioms to the type system provided they also supply a decision procedure for the richer logic. In Section 5.2, we show how to add security policy-specific axioms that allow many unnecessary security checks to be eliminated.

Types As discussed in the previous section, we reflect values into the type structure using the singleton types $b(P)$ (where b is a base type and P has kind `Val`) and $\mathtt{S}(P)$ (where P has kind `State`).

The second main type constructor is a polymorphic function type $\forall[\Delta].(P, \tau_1, \ldots, \tau_n) \to \mathtt{0}$. The predicate context Δ abstracts a series of predicate variables and requires that a sequence of boolean-valued predicates be satisfied before the function can be invoked. The predicate P in the first argument position is not a run-time argument to the function. Rather, it is another precondition requiring that the function be called in the state associated with P. The actual function arguments must have types τ_1 through τ_n.

The notation "$\to\mathtt{0}$" at the end of each function type is intended to indicate that functions in our language do not "return" the way high-level language functions normally do. All functions are written in continuation-passing style (CPS) so they are passed their return address (another function) explicitly and "return" by calling that function. We have chosen to present the language in CPS for two reasons. First, CPS linearizes the code and makes the flow of control evident; this is convenient for the type system because it must thread the static state component along the control-flow path. Second, CPS makes all run-time control transfers and compile-time information transfers occur using the same uniform mechanism. In the latter case, code can use the same state abstraction mechanism (polymorphism) at all join points in the code (function returns, if statements) as it does on function call. Propagation of information from one code block to the next also occurs uniformly in all cases by proving a function pre-condition. It is possible to avoid using CPS, but then we would need some special syntax to indicate how to propagate information from the two branches of an if statement to the join point, for instance. Of course, there are other reasons why CPS may be inconvenient in an implementation, and if this is so, the implementation could certainly add some special syntax.

We specify the well-formedness of types using the judgement $\Phi \vdash \tau$. In general, a type is well-formed if Δ contains the free variables of the type. Function types of the form $\forall[\Delta].(P, \tau_1, \ldots, \tau_n) \to \mathtt{0}$ also require that P has kind `State` and that the predicates occuring in Δ have kind $\mathcal{B}$. These rules are straight-forward, and due to space considerations, we have omitted them. They may be found in the technical report [29]. Because predicates are uninterpreted, we can use standard syntactic equality of types up to α-conversion of bound variables.

Values and Expressions The typing rules for values have the form $\Phi \vdash v : \tau$ and state that the value v has type τ in the given context. The judgement $\Phi \vdash e$ states that the expression e is well-formed. Recall that CPS expressions do not return values, and hence the latter judgement is not annotated with a return type. Figure 6 presents the formal rules. In these judgements, we use the notation $\Phi, \rho{:}\kappa$ to denote a new context in which the binding $\rho{:}\kappa$ has been appended to the list of assumptions in Φ. The operation is undefined if ρ appears in Φ. The notations Φ, P and $\Phi, x{:}\tau$ and the extension to Φ, Δ are similar, although P may already appear in Φ.

The values include variables and constants. The treatment of variables is standard, and a signature $\mathcal{C}$ gives types to the constants.

The value $v[P]$ is the instantiation of the polymorphic value v with the predicate P. We consider this instantiation a value because predicates are used only for type-checking purposes; they have no run-time significance. The value $v[\cdot]$ is somewhat similar: If v has type $\forall[P, \Delta].(\cdots) \rightarrow \mathtt{0}$ and we can prove the predicate P is valid in the current context, we give $v[\cdot]$ the type $\forall[\Delta].(\cdots) \rightarrow \mathtt{0}$. Again, the notation $[\cdot]$ is used only to specify that the type-checker should attempt to prove the pre-condition; it will not influence the execution of programs. In a system with a more sophisticated logic than the one presented in this paper, we might not want to trust the correctness of complex decision procedures for the logic. In this case, we would replace $[\cdot]$ with a proof of the pre-condition and replace the type checker's decision procedure with a much simpler proof-checker.

Target language function values specify a list of pre-conditions using the predicate context Δ. Every function also expresses a state pre-condition P. The function can only be called in the state denoted by P. Static semantics rule (10) contains the the judgement $\Phi \vdash P$ in_state, which ensures this invariant is maintained. This rule also enforces the standard constraints that argument types must match the types of the formal parameters. Finally, because the predicate context is empty, any pre-conditions the function might have specified must have already been proven valid.

Rule (6) states that we type check the body of a function assuming its pre-conditions hold. In this rule, we use the notation $\Phi \leftarrow P$ to denote a context Φ' in which the state component of Φ has been replaced by P. For example, suppose a function g is defined in the context $\Delta'; \Gamma'; P'$. The type checker can use any of the predicates in Δ' to help prove g is well-formed but it cannot assume that g will be called in the state P'. The function g is defined here, but may not be used until much later in the computation when the state is different (P'' perhaps).

It is tempting to define a predicate "$in\_state(P)$" and to include this predicate in the list of function pre-conditions Δ. Using this mechanism, it may appear as though we could eliminate the special-purpose state component of the type-checking context. Unfortunately, this simplification is unsound. Consider the following informal example:

% Assume current state = $start$
let $g{:}\forall[in\_state(start)].(\tau_1, \ldots, \tau_n) \rightarrow \mathtt{0} = \cdots$ in
% Prove pre-condition:
let $g'{:}\forall[\,].(\tau_1, \ldots, \tau_n) \rightarrow \mathtt{0} = g[\cdot]$ in
% Change the state to q' where $q' \neq start$:
let $\_ = op()$ in
% g is not called in the $start$ state!
$g'(v_1, \ldots, v_n)$

On the last line, the function g is invoked in a state q' when the function definition assumed it would be invoked in the *start* state. The example highlights the main difference between the state predicates and the others: The validity of the predicates in Δ is invariant throughout the execution of the program whereas the validity of a state predicate varies during execution because it depends implicitly on the current state of the machine.

Existential values are handled in standard fashion [14]. The value $\mathtt{pack}[P, v]$ *as* $\exists \rho{:}\kappa.\tau$ creates an existential package that hides P in τ using ρ. The corresponding elimination form, $\mathtt{let}\,\rho, x = \mathtt{unpack}\, v'\, \mathtt{in}\, e$ unpacks the existential v', substituting v for x and P for ρ into the remaining expression e. As with polymorphic types, we assume a type-erasure interpretation of existentials.

Finally, the conditional $\mathtt{if}\, v\, (q \rightarrow e_1 \mid \_ \rightarrow e_2)$ tests an automaton state v to determine whether v is the state q or not. If so, the program executes e_1 and if not, the program executes e_2. This expression is a variant of Harper and Morrisett's typecase operator [6], and like typecase, it performs type refinement. If v has type $\mathtt{S}(\rho)$ then $\mathtt{if}$ refines the type-checking context for e_1 with the information that $\rho = \hat{q}$ by substituting $\hat{q}$ for ρ. It refines the type-checking context for e_2 with the information $\neq (\rho, \hat{q})$.[2] Programs can use this mechanism to dynamically check whether or not they are about to enter the bad state, and if not, they can present evidence of this fact to the type system.

There is no need to use the $\mathtt{if}\, v$ construct if we know which state a value v represents. For example, we know the expression $\mathtt{if}\, q\, (q \rightarrow e_1 \mid \_ \rightarrow e_2)$ will reduce to e_1 and therefore e_2 is dead code and the test is wasted computation. However, during the proof of soundness of the type system, such configurations arise and cause difficulties. To avoid these difficulties, we follow the strategy of Crary *et al.* [1] and add the *trivialization* rules (14) and (15) which deal with these redundant cases. Each rule type checks only the branch of the if statement that will be taken.

Operational Semantics The operational semantics for the language is given by the relation $e \longmapsto_s e'$ (see Figure 7). The symbol s is either empty $(\cdot)$ or it is a protected function applied to some number of arguments $(f(a_1, \ldots, a_n))$. Most operations, and, in fact, all of the operations shown in Figure 7, emit the empty symbol. However, this figure does not show the operation of the protected functions. In the next section, we will explain the operational semantics of the protected functions f in the context of a signature for a particular security automaton.

We have given a typed operational semantics to facilitate the proof of soundness of the system. However, inspection of the rules will reveal that evaluation does not depend upon types or predicates, provided the expressions are well-formed. Therefore we can type-check a program and then erase the types before executing it.

4.2 The Secure System Interface

In order to specialize the generic language, we construct a typed interface or *signature* for the constants in the language. Untrusted code will link against this interface. However, recall from the introduction that application programmers are intended to follow their regular routine of programmming in a high-level language against a high-level system interface. Therefore, we need to obtain the secure target language interface from the high-level interface and the security automaton definition.

We assume that the high-level interface for constants a and protected operations f is given by a signature $\mathcal{C}_{source}$

[2] Another reasonable choice would be to add an equality predicate $= (\rho, \hat{q})$ to the context on a successful branch instead of performing a substitution. This would make checking the two branches more symmetric. However, it would be necessary to look up these equality predicates to decide type equality. By using substitution, we may continue to use simpler syntactic equality on types.

$$v(v_1, \ldots, v_n) \longmapsto. \; e_m[v, v_1, \ldots, v_n/g, x_1, \ldots, x_n] \qquad \begin{array}{l} \text{if } v = v'\,\phi_1 \cdots \phi_m \\ \text{and } v' = \texttt{fix}\,g[\theta_1, \ldots, \theta_m].(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e_0 \\ \text{and for } 1 \le i \le m, \\ \quad \phi_i = [\cdot] \text{ and } \theta_i = P_i \text{ and } e_i = e_{i-1}, \text{ or} \\ \quad \phi_i = [P_i'] \text{ and } \theta_i = \rho_i{:}\kappa_i \text{ and } e_i = e_{i-1}[P_i'/\rho_i] \end{array}$$

$$\texttt{let}\,\rho, x = \texttt{unpack}\,(\texttt{pack}[P, v] \; as \; \tau)\,\texttt{in}\,e \longmapsto. \; e[P, v/\rho, x]$$

$$\texttt{if}\,q'\;(q \to e_1 \mid \_ \to e_2) \longmapsto. \; e_1 \qquad \text{if } q' = q$$

$$\texttt{if}\,q'\;(q \to e_1 \mid \_ \to e_2) \longmapsto. \; e_2 \qquad \text{if } q' \ne q$$

Figure 7: Target Language Operational Semantics

where $\mathcal{C}_{source}(a)$ is some base type b and $\mathcal{C}_{source}(f)$ is some function type taking a single continuation $(b_1, \ldots, b_n, (b) \to \texttt{0}) \to \texttt{0}$. A security automaton $\mathcal{A}$ defines the operations $f(a_1, \ldots, a_n)$ that are allowed.[3] Together, the automaton $\mathcal{A}$ and signature $\mathcal{C}_{source}$ are use to define the target language interface. This definition is presented in Figure 8 and has three parts: the type signature $\mathcal{I}$, the value signature $\mathcal{C}$, and the operational signature.

The type signature gives each constant $\hat{a}$ and automaton state $\hat{q}$ kinds `Val` and `State` respectively. Furthermore, for each protected function f, the type signature specifies a predicate δ_f. The type system ensures that for all $\hat{q_1}, \hat{q_2}$, $\hat{a}_1, \ldots, \hat{a}_n$, we will be able to prove $\delta_f(\hat{q_1}, \hat{q_2}, \hat{a}_1, \ldots, \hat{a}_n)$ only if the corresponding automaton transition holds. In other words, only if $\delta(f)(q_1, a_1, \ldots, a_n) = q_2$

The value signature specifies that objects a and states q are given the correct singleton types. For each protected function f in the source language, the target language contains two functions. The function δ_f is supplied by the implementer of the security policy; it is used to dynamically determine the automaton state transition function given the program executes the function f in the state ϱ_1 with arguments identified by $\rho_1, \ldots, \rho_n$. When δ_f has computed the transition, it calls its continuation, passing it the next state ϱ_2 so the continuation can test this state to determine whether it is *bad*. The continuation assumes the predicate $\delta_f(\varrho_1, \varrho_2, \rho_1, \ldots, \rho_n)$. Before calling the function f itself, we require that the type-checker be able to prove that f will make a transition to some state other than the bad state. Hence the pre-condition on f states that we must know the automaton transition that will occur $(\delta_f(\varrho_1, \varrho_2, \rho_1, \ldots, \rho_n))$ and moreover that the new state ϱ_2 is not equal to *bad*.

Now that we have fully defined the target language and specialized it for a particular security automaton, we can prove that the type system satisfies a number of properties. In the next section, we show the target language type system is sound and enforces the security policy.

4.3 Properties of the Secure Target Language

A predicate P is valid with respect to an automaton $\mathcal{A}$, written $\mathcal{A} \models P$, if:

- P is $\ne(\hat{q}, \hat{q}')$ and $q \ne q'$, or
- P is $\delta_f(\hat{q_1}, \hat{q_2}, \hat{a}_1, \ldots, \hat{a}_n)$ and $\delta(f)(q_1, a_1, \ldots, a_n) = q_2$

We say that an expression e is *secure* with respect to a security automaton $\mathcal{A}$ in state q, written $\mathcal{A}; q \vdash e$, if

1. $q \ne bad$

and there exist predicates $P_1, \ldots, P_n$ such that:

2. $\mathcal{A} \models P_i$, for $1 \le i \le n$
3. $P_1, \ldots, P_n; \cdot; \hat{q} \vdash e$

If we can prove that an expression e is secure in our deductive system then the expression should not violate the security policy when it executes. The soundness theorem below formalizes this notion. The first part of the theorem, *Type Soundness*, ensures that programs obey a basic level of control-flow safety. More specifically, it ensures that expressions do not get *stuck* during the course of evaluation. An expression e is *stuck* if e is not `halt` and there does not exist an e' such that $e \longmapsto_s e'$. Hence, Type Soundness implies a program will only halt when it executes the halt instruction and not because we have applied a function to too few or the wrong types of arguments. The second part of the theorem, *Security*, ensures programs obey the policy specified by the security automaton $\mathcal{A}$. In other words, the sequence of protected operations executed by the program must form a string in the language $\mathcal{L}(\mathcal{A})$. In this second statement, we use the notation $|s_1, \ldots, s_n|$ to denote the subsequence of the symbols $s_1, \ldots, s_n$ with all occurences of $\cdot$ removed.

Theorem 1 (Soundness)
If $\mathcal{A}; q_0 \vdash e_1$ *then*

1. (Type Soundness) *For all evaluation sequences* $e_1 \longmapsto_{s_1} e_2 \longmapsto_{s_2} \cdots \longmapsto_{s_n} e_{n+1}$, *the expression* e_{n+1} *is not* stuck.
2. (Security) *If* $e_1 \longmapsto_{s_1} e_2 \longmapsto_{s_2} \cdots \longmapsto_{s_n} e_{n+1}$ *then* $|s_1, s_2, \ldots, s_n| \in \mathcal{L}(\mathcal{A})$

Soundness can be proven syntactically in the style of Wright and Felleisen [30] using the following two lemmas. The proof appears in a companion technical report [29].

Lemma 2 (Progress) *If* $\mathcal{A}; q \vdash e$ *then either:*

1. $e \longmapsto_s e'$ *or*
2. $e = \texttt{halt}$

[3] Notice that the automaton does not use the continuation to decide whether or not to allow the protected operation. For simplicity, automaton functions depend exclusively on objects of base type. See Section 4.4 for more explanation.

Type and Value Signature

$$
\begin{array}{lcl}
\mathcal{I}(\hat{a}) & = & \texttt{Val} \qquad \text{for } a \in \mathrm{A} \\
\mathcal{I}(\hat{q}) & = & \texttt{State} \qquad \text{for } q \in Q \\
\mathcal{I}(\delta_f) & = & (\texttt{State}, \texttt{State}, \overbrace{\texttt{Val}, \ldots, \texttt{Val}}^{n}) \to \mathcal{B} \\
& & \quad \text{if } \mathcal{C}_{source}(f) = (b_1, \ldots, b_n, (b) \to 0) \to 0 \\
\\
\mathcal{C}(a) & = & b(\hat{a}) \qquad \text{if } \mathcal{C}_{source}(a) = b \\
\mathcal{C}(q) & = & \mathtt{S}(\hat{q}) \qquad \text{if } q \in Q \\
\mathcal{C}(\delta_f) & = & \forall[\varrho_1{:}\texttt{State}, \rho_1{:}\texttt{Val}, \ldots, \rho_n{:}\texttt{Val}, \neq (\varrho_1, bad)].(\varrho_1, S(\varrho_1), b_1(\rho_1), \ldots, b_n(\rho_n), \\
& & \quad \forall[\varrho_2{:}\texttt{State}, \delta_f(\varrho_1, \varrho_2, \rho_1, \ldots, \rho_n)](\varrho_1, S(\varrho_2)) \to 0) \to 0 \\
& & \quad \text{if } \mathcal{C}_{source}(f) = (b_1, \ldots, b_n, (b) \to 0) \to 0 \\
\mathcal{C}(f) & = & \forall[\varrho_1{:}\texttt{State}, \varrho_2{:}\texttt{State}, \rho_1{:}\texttt{Val}, \ldots, \rho_n{:}\texttt{Val}, \neq (\varrho_2, bad), \delta_f(\varrho_1, \varrho_2, \rho_1, \ldots, \rho_n)]. \\
& & \quad (\varrho_1, b_1(\rho_1), \ldots, b_n(\rho_n), \forall[].(\varrho_2, \exists\rho{:}\texttt{Val}.b(\rho)) \to 0) \to 0 \\
& & \quad \text{if } \mathcal{C}_{source}(f) = (b_1, \ldots, b_n, (b) \to 0) \to 0
\end{array}
$$

Operational Signature

$$
\begin{array}{ll}
\delta_f[\hat{q_1}][\hat{a_1}] \cdots [\hat{a_n}][\cdot](q_1, a_1, \ldots, a_n, v_{cont}) & \text{if } \delta(f)(q_1, a_1, \ldots, a_n) = q_2 \\
\quad \longmapsto_{\cdot} v_{cont}[\hat{q_2}][\cdot](q_2) & \text{and for } 1 \leq i \leq n, \mathcal{C}_{source}(a_i) = b_i \\
& \text{and } \mathcal{C}_{source}(f) = (b_1, \ldots, b_n, (b) \to 0) \to 0 \\
\\
f[\hat{q_1}][\hat{q_2}][\hat{a_1}] \cdots [\hat{a_n}][\cdot][\cdot](a_1, \ldots, a_n, v_{cont}) & \text{if for } 1 \leq i \leq n, \mathcal{C}_{source}(a_i) = b_i \\
\quad \longmapsto_{f(a_1, \ldots, a_n)} v_{cont}(\texttt{pack}[\hat{a}, a] \textit{ as } \exists\rho{:}\texttt{Val}.b(\rho)) & \text{and } \mathcal{C}_{source}(f) = (b_1, \ldots, b_n, (b) \to 0) \to 0 \\
\quad (\text{for some } a \text{ such that } \mathcal{C}_{source}(a) = b) &
\end{array}
$$

Figure 8: Target Language Interface

Lemma 3 (Subject Reduction) *If* $\mathcal{A}; q \vdash e$ *and* $e \longmapsto_s e'$ *then*

1. *if* $s = \cdot$ *then* $\mathcal{A}; q \vdash e'$
2. *and if* $s = f(a_1, \ldots, a_n)$ *then* $\mathcal{A}; q' \vdash e'$ *where* $\delta(f)(q, a_1, \ldots, a_n) = q'$

Finally, inspection of the typing rules will reveal that for any expression or value, there is exactly one typing rule that applies and that the pre-conditions for the rules only depend upon subcomponents of the terms or values (with possibly a predicate substitution). Judgements for the well-formedness of types and predicates are also well-founded so the type system is decidable:

Proposition 4 *It is decidable whether or not* $\Phi \vdash e$.

4.4 Discussion

The type system described in the previous section is somewhat complicated. Fortunately, little of the type system is specialized to encode security automata in particular. For example, the basic building blocks of the language are polymorphism, singleton types, and existential types. Strongly-typed languages for optimizing data representations like Xi and Pfenning's Dependent ML [33] already contain these type constructors so that they can safely solve other data representation problems. For example, singleton types are useful for representing tagged unions [15] and performing array bounds check elimination [32]. The TIL(T) and FLINT compilers use existential types to encode closures [13] and Java classes [9], respectively. Polymorphism is ubiquitous.

On the other hand, in order to represent the security automaton's state and its transition function, the typing rules for the language manipulate a state component and a collection of predicates, which are less standard features for a type system. Still, these features do appear in other type systems for low-level languages. For instance, TALx86 [15], the Typed Assembly Language implementation, contains a state component for reasoning about aliasing and stateful operations such as object deallocation [26]. Both Dependent ML and TALx86 contain a collection of predicates for reasoning about arithmetic so that programs can eliminate array bounds checks.

The fact that there can be significant reuse of many of the type constructors in our language makes the trusted computing base more manageable. It also eases the implementation burden. We believe it is feasible to add all of the features necessary to scale the language up to a full, practical compiler intermediate or target language. In fact, as noted above, the TALx86 implementation already contains almost all of the necessary features. Below, we briefly mention a few of the issues.

Language Extensions In general, adding standard (type-safe) programming language features for use by the application program poses no difficulties for this framework. However, if security policies themselves depend upon higher-order functions or immutable data structures such as tuples and records, we will have to track the values of these data structures in the type system using singleton types as we did for values of base type. The simplest way to handle this extension is to use an allocation semantics (See, for example, [16]). In this setting, when a function closure $\texttt{fix}\, g[\Delta](\cdots).e$ is allocated, it is bound to a new address (ℓ). Instead of substituting the closure through the rest of the code as we do now, we would substitute the address (ℓ) through the code

$$
\begin{array}{lcl}
|b| & = & \exists\rho{:}\mathtt{Val}.b(\rho) \\
|(\tau_1,\ldots,\tau_n)\rightarrow 0| & = & \forall[\varrho{:}\mathtt{State}, \neq(\varrho, bad)].(\varrho, \mathtt{S}(\varrho), |\tau_1|,\ldots,|\tau_n|)\rightarrow 0 \\
\\
|x| & = & x \\
|a| & = & \mathtt{pack}[\hat{a}, a]\ as\ \exists\rho{:}\mathtt{Val}.b(\rho) \quad \text{if } \mathcal{C}_{source}(a) = b \\
|\mathtt{fix}\,g(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e| & = & \mathtt{fix}\,g[\varrho{:}\mathtt{State},\neq(\varrho,bad)].(\varrho, x{:}S(\varrho), x_1{:}|\tau_1|,\ldots,x_n{:}|\tau_n|).|e|_{\varrho,x} \\
|f| & = & \mathtt{fix}\,\_[\varrho_1{:}\mathtt{State},\neq(\varrho_1,bad)](\varrho_1, x_0{:}S(\varrho_1), x_1{:}|b_1|,\ldots,x_n{:}|b_n|, x_{n+1}{:}|(b)\rightarrow 0|). \\
 & & \quad \mathtt{let}\,\rho_1, x_1' = \mathtt{unpack}\,x_1\,\mathtt{in} \\
 & & \quad \ldots \\
 & & \quad \mathtt{let}\,\rho_n, x_n' = \mathtt{unpack}\,x_n\,\mathtt{in} \\
 & & \quad \mathtt{let}\,\varrho_2, \delta_f(\varrho_1,\varrho_2,\rho_1,\ldots,\rho_n), x_{\varrho_2} = \delta_f[\varrho_1][\rho_1]\cdots[\rho_n][\cdot](x_0, x_1',\ldots,x_n')\,\mathtt{in} \\
 & & \quad \mathtt{if}\,x_{\varrho_2}(\\
 & & \qquad bad \rightarrow \mathtt{halt} \\
 & & \qquad |\ \_ \rightarrow \mathtt{let}\,x = f[\varrho_1][\varrho_2][\rho_1]\cdots[\rho_n][\cdot][\cdot](x_1',\ldots,x_n')\,\mathtt{in} \\
 & & \qquad\quad x_{n+1}[\varrho_2][\cdot](x_{\varrho_2}, x)) \\
 & & \text{if } \mathcal{C}_{source}(f) = (b_1,\ldots,b_n,(b)\rightarrow 0)\rightarrow 0 \\
\\
|v_0(v_1,\ldots,v_n)|_{P,v} & = & |v_0|[P][\cdot](v, |v_1|,\ldots,|v_n|) \\
|\mathtt{halt}|_{P,v} & = & \mathtt{halt}
\end{array}
$$

Figure 9: Program Instrumentation

$$
\begin{array}{llcl}
\textit{types} & \tau & ::= & b \mid (\tau_1,\ldots,\tau_n)\rightarrow 0 \\
\textit{constants} & a & \in & \mathrm{A} \\
\quad\textit{protected ops} & f & \in & \mathrm{F} \\
\textit{values} & v & ::= & x \mid a \mid f \mid \\
 & & & \mathtt{fix}\,g(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e \\
\textit{expressions} & e & ::= & v_0(v_1,\ldots,v_n) \mid \mathtt{halt}
\end{array}
$$

Figure 10: Source Language Syntax

and give it the singleton type $\tau(\hat{\ell})$ where τ is $\forall[\Delta](\cdots)\rightarrow 0$. All the other mechanisms remain unchanged. For the sake of simplicity, we decided not to present this style of semantics.

Adding mutable data structures to the programming language is also straight-forward, unless the security policies depend upon their contents. In the latter case, the state of that data must be encoded in the state of the automaton. The assignment operator must be designated as a protected operation that changes the state.

There are also several ways to extend the type portion of the language. For example, Xi and Pfenning's Dependent ML contains existential dependent types of the form $\exists[\rho{:}\mathtt{Val}, P(\rho)].\tau(\rho)$, which can be read "there exists a value ρ such that $P(\rho)$ and that ρ has type τ." Such existentials could be useful in a security setting. For example, they would allow users to manipulate collections of files that all have the property that they are readable or writeable.

5 Program Instrumentation

It is easy to design a translation that instruments a safe source language program with security checks now that we have set up the appropriate type-theoretic machinery in the target language. In this section, we instrument programs written in a simply-typed continuation-passing style language. The syntax of the language can be found in Figure 10. We will assume a static semantics given by judgements $\Gamma \vdash_{source} v : \tau$ and $\Gamma \vdash_{source} e$ where Γ is a finite map from value variables to types. Constants a and f are given types by the source signature $\mathcal{C}_{source}$ described in the previous section. Other than this, the semantics are entirely standard and have been omitted. Figure 9 gives the instrumentation algorithm in two parts: a type translation and a term translation. Here and in later sections, we will use the abbreviation:

$$
\begin{array}{l}
\mathtt{let}\,\Delta, x_1,\ldots,x_n = v(v_1,\ldots,v_n)\,\mathtt{in}\,e \equiv \\
v(v_1,\ldots,v_n,\mathtt{fix}\,\_[\Delta].(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e)
\end{array}
$$

In the translation in Figure 9, we assume predicate and value variables bound by `let` are fresh.

The interesting portion of the type translation involves the translation of function types. The static semantics maintains the invariant that programs never enter the bad state and we naturally express this fact as a pre-condition to every function call. Hence the translation of $(\tau_1,\ldots,\tau_n)\rightarrow 0$ is $\forall[\varrho{:}\mathtt{State},\neq(\varrho,bad)].(\varrho,\mathtt{S}(\varrho),|\tau_1|,\ldots,|\tau_n|)\rightarrow 0$. In general, we may not know the current state statically so we quantify over all states ϱ, provided $\varrho \neq bad$. In order to determine state transfers do not go wrong, we will also have to thread a representation of the state ($\mathtt{S}(\varrho)$) through the computation.

Every value of base type b is packed up as the existential $\exists\rho{:}\mathtt{Val}.b(\rho)$ so that it can be used generically in the normal case. In other words, wherever we used an integer in the source language, we will be able to use the target language type $\exists\rho{:}\mathtt{Val}.int(\rho)$. However, when we come to the translation of a protected operation, these generic values will be unpacked so the type system can maintain precise information about them. In fact, examining Figure 9, we can see that the first step in the translation of protected operations f is to unpack its arguments. Next, we use the function δ_f to determine the state transition that will occur if we execute f on these arguments. After checking to ensure we do not enter the bad state, we execute f itself passing it a continuation that executes in state ϱ_2.

Instrumented programs type check and thus they are *secure* in the sense made precise in the last section:

Proposition 5 *If* $\vdash_{source} e$ *then* $\mathcal{A}; q_0 \vdash |e|_{\hat{q_0},q_0}$.

This property can be proven using a straightforward induction on the typing derivation of the source term.

5.1 An Example: The Taxation Applet

In order to demonstrate the translation, we have written a simple "taxation applet" in the source language. When invoked, this applet sends a request out for tax forms. After sending the request, the applet reads a private file containing the customer salary before computing the taxes owed. We will assume files ($file$) and integers (int) are available as base types; $send$ and $read$ are the two protected operations:

$$
\begin{array}{ll}
\texttt{fix}\,\_(secret{:}file, x_{cont}{:}(int) \rightarrow \texttt{0}). & \\
\quad \texttt{let}\,\_ \quad\;\; = send()\,\texttt{in} & \texttt{\% send for tax forms} \\
\quad \texttt{let}\,salary = read(secret)\,\texttt{in} & \texttt{\% read salary} \\
\quad \texttt{let}\,taxes \;\; = salary\,\texttt{in} & \texttt{\% compute taxes!!} \\
\quad x_{cont}(taxes) &
\end{array}
$$

The code below shows the results of instrumentating the taxation applet with checks from the simple file system security automaton of Section 3. We have simplified the output of the formal translation slightly to make it more readable. In particular, we have inlined the functions that the translation wraps around each protected operation. When reading the code calling $send$ or $read$, notice that the instantiation of predicate variables indicates the state transition that occurs. For example, execution of the expression $send[\varrho_1][\varrho_2][\cdot][\cdot]()$ causes the automaton to make a transition from ϱ_1 to ϱ_2. When reading the checking functions, for example, $\delta_{send}[\varrho_1][\cdot](x_{\varrho_1})$, notice that we are checking the validity of the operation in the state indicated by the arguments (ϱ_1 and x_{ϱ_1}) and that the result is the next state.

$$
\begin{array}{l}
\texttt{fix}\,\_[\varrho_1{:}\texttt{State}, \neq(\varrho_1, bad)] \\
\qquad (\varrho_1, x_{\varrho_1}{:}\texttt{S}(\varrho_1), secret{:}\exists\rho{:}\texttt{Val}.file(\rho), \\
\qquad x_{cont}{:}\tau_{cont}). \\
\quad \texttt{let}\,\varrho_2, \delta_{send}(\varrho_1, \varrho_2), x_{\varrho_2} = \delta_{send}[\varrho_1][\cdot](x_{\varrho_1})\,\texttt{in} \\
\quad \texttt{if}\,x_{\varrho_2}\,(\\
\qquad bad \rightarrow \texttt{halt} \\
\qquad \mid \_\!\rightarrow \\
\qquad\quad \texttt{let}\,\_ = send[\varrho_1][\varrho_2][\cdot][\cdot]()\,\texttt{in} \\
\qquad\quad \texttt{let}\,\rho, secret' = \texttt{unpack}\,secret\,\texttt{in} \\
\qquad\quad \texttt{let}\,\varrho_3, \delta_{read}(\varrho_2, \varrho_3, \rho), x_{\varrho_3} = \\
\qquad\qquad \delta_{read}[\varrho_2][\cdot](x_{\varrho_2}, secret')\,\texttt{in} \\
\qquad\quad \texttt{if}\,x_{\varrho_3}\,(\\
\qquad\qquad bad \rightarrow \texttt{halt} \\
\qquad\qquad \mid \_\!\rightarrow \\
\qquad\qquad\quad \texttt{let}\,salary = read[\varrho_2][\varrho_3][\cdot][\cdot](secret')\,\texttt{in} \\
\qquad\qquad\quad \texttt{let}\,taxes = salary\,\texttt{in} \\
\qquad\qquad\quad x_{cont}[\varrho_3][\cdot](x_{\varrho_3}, taxes)))
\end{array}
$$

where $\tau_{cont} =$

$$
\begin{array}{l}
\quad \forall[\varrho_{cont}, \neq(\varrho_{cont}, bad)]. \\
\qquad (\varrho_{cont}, \texttt{S}(\varrho_{cont}), \exists\rho'{:}\texttt{Val}.int(\rho')) \rightarrow \texttt{0}
\end{array}
$$

The translation does not assume that the taxation applet is invoked in the initial automaton state and consequently the resulting function abstracts the input state ϱ_1. Also, as specified by the translation, objects of base type, like the file $secret$ become existentials. The main point of interest in this example is that before each of the protected operations $send$ and $read$, the corresponding automaton function determines the next state. Then the `if` construct checks that these states are not bad. In the successful branch, the type checker introduces information into the context that allows it to infer that executing the $read$ and $send$ operations is safe.

5.2 Optimization

Many security automata exhibit special structure that allows us to optimize secure programs by eliminating checks that are inserted by the naive program instrumentation procedure [27]. One common case is an operation f that always succeeds in a given state q and transfers control to a new state q' regardless of its arguments. In this situation, we can make the following axiom available to the type checker:

$$\frac{}{\Phi \vdash \delta_f(q, q', P_1, \ldots, P_n)} \text{ (for all } P_1, \ldots, P_n)$$

If we know we are in state q, we can use the axiom above and the fact that $q \neq bad$ to satisfy the pre-condition on f; there is no need to perform a run-time check.

The $send$ operation in the security automaton in Section 3 has this property. When invoked in the initial state, $send$ always succeeds and execution continues in the initial state. Therefore, we can safely add the axiom:

$$\frac{}{\Phi \vdash \delta_{send}(start, start)}$$

Now, if we know our taxation function is only invoked in the $start$ state, we can rewrite it, eliminating one of the run-time checks:

$$
\begin{array}{l}
\texttt{\% Optimization 1:} \\
\texttt{fix}\,\_[\,](start, x_{start}{:}\texttt{S}(start), secret{:}\exists\rho{:}\texttt{Val}.file(\rho), \\
\qquad x_{cont}{:}\tau_{cont}). \\
\quad \texttt{let}\,\_ = send[start][start][\cdot][\cdot]()\,\texttt{in} \\
\quad \ldots
\end{array}
$$

The type checker can prove $send$ is executed in the $start$ state and that the predicate $\delta_{send}(start, start)$ and the predicate $\neq(start, bad)$ are valid. Therefore, the optimized applet continues to type-check.

A second important way to optimize target language programs is to perform a control-flow analysis that propagates provable predicates statically through the program text. Using this technique, we can further optimize the taxation applet. Assume the calling context can prove the predicate $\delta_{read}(start, has\_read, \rho)$ (perhaps a run-time check was performed at some earlier time) where ρ is the value predicate corresponding to the file $secret$. In this case, the caller can invoke a tax applet with a stronger pre-condition that includes the predicate $\delta_{read}(start, has\_read, \rho)$. Moreover, with this additional information, an optimizer can eliminate the redundant check surrounding the file read operation:

$$
\begin{array}{l}
\texttt{\% Optimization 2:} \\
\texttt{fix}\,\_[\rho{:}\texttt{Val}, \delta_{read}(start, has\_read, \rho)](start, \\
\qquad secret{:}file(\rho), \\
\qquad x_{cont}{:}\forall[\,].(has\_read, \exists\rho{:}\texttt{Val}.int(\rho)) \rightarrow \texttt{0}). \\
\quad \texttt{let}\,\_ = send[start][start][\cdot][\cdot]()\,\texttt{in} \\
\quad \texttt{let}\,salary = read[start][has\_read][\cdot][\cdot](secret)\,\texttt{in} \\
\quad \texttt{let}\,taxes = salary\,\texttt{in} \\
\quad x_{cont}(taxes)
\end{array}
$$

In the code above, the optimizer rewrites the applet precondition with the necessary information. The caller is now obligated to prove the additional pre-condition before the applet can be invoked. The caller also unpacks the secret file before making the call so that the type checker can make the connection between the arguments to the δ_{read} predicate and this particular file. Finally, because the automaton state transitions are statically known throughout this program, we do not need to thread the state representation through the program. We assumed an optimizer was able to detect this unused argument and eliminate it. After performing all these optimizations, the resulting code is operationally equivalent to the original taxation applet from section 5.1, but provably secure.

The flexibility in the type system is particularly useful when a program repeatedly performs the same restricted operations. A more sophisticated tax applet might need to make a series of reads from the secret file (for charitable donations, number of dependents, etc.). If we assume the recursive function *read_a_lot* performs these additional reads, we need no additional security checks:

$$
\begin{array}{l}
\texttt{fix}\, read\_a\_lot \\
\quad [\varrho{:}\texttt{State}, \rho{:}\texttt{Val}, \delta_{read}(\varrho, has\_read, \rho), \neq (\varrho, bad)] \\
\quad (\varrho, secret{:}file(\rho), \\
\quad x_{cont}{:}\forall[\,](has\_read, \exists\rho{:}\texttt{Val}.int(\rho)) \rightarrow \texttt{0}). \\
\texttt{\% In unknown state } \varrho \\
\texttt{let}\, info = read[\varrho][has\_read][\cdot][\cdot]()\,\texttt{in} \\
\texttt{\% In known state } has\_read \\
\cdots \\
\texttt{\% Must prove } \delta(has\_read, has\_read, \rho) \\
read\_a\_lot[has\_read][\rho][\cdot][\cdot](secret, x_{cont})
\end{array}
$$

The *read_a_lot* function can be invoked in a good state ϱ (*i.e.* either *start* or *has_read*) when we can prove the predicate $\delta_{read}(\varrho, has\_read, \rho)$. Using the δ_{read} predicate in the function pre-condition, the type checker infers that the read operation transfers control from the ϱ state to the *has_read* state. Before the recursive call, the type checker has the obligation to prove $\delta_{read}(has\_read, has\_read, \rho)$ but it cannot do so because it only knows that $\delta_{read}(\varrho, has\_read, \rho)$! Fortunately, we can remedy this problem by adding another policy-specific rule to the type-checker:

$$
\frac{\begin{array}{c}\Phi \vdash \neq (P, bad) \\ \Phi \vdash \delta_{read}(P, has\_read, P_f)\end{array}}{\Phi \vdash \delta_{read}(P', has\_read, P_f)} \quad (\text{for all } P, P', P_f)
$$

This rule states that if we can read a file P_f in one state (P), then we can read it in any state (except the *bad* one) and we always move to the *has_read* state. This condition is easily decidable.

In practice, Erlingsson and Schneider's untyped optimizer analyzes security automaton structure and performs optimizations similar to the ones discussed above. Once the optimizer has obtained the information necessary for a particular transformation, this information can also be used to automatically generate the policy-specific axioms that we have discussed.

6 Related Work

The design of the target language was inspired by Xi and Pfenning's Dependent ML (DML) [33, 31]. As in DML, we track the identity of values using dependent refinement types and singleton types. However, rather than applying the technology to array bounds check elimination and dead-code elimination, we have applied it to the problem of expressing security policies.

Leroy and Rouaix [11] also consider security in the context of strongly-typed languages. Their main concern is proving that standard strongly-typed languages provide certain security properties. For example, they show that a program written in a typed lambda calculus augmented with references cannot modify unreachable (in the sense of tracing garbage collection) locations. They did not investigate mechanisms for mechanically checking that instrumented programs are safe, nor did they study the broader range of security policies that can be specified using security automata.

Using code instrumentation to enforce safety properties is not new. Software Fault Isolation (SFI) [28] rewrites binaries to ensure control-flow and memory safety properties. However, SFI cannot enforce the broad set of policies that can be specified by security automata.

Other systems enforce a very rich collection safety properties using code instrumentation and other techniques. For instance, Evans and Twyman [2] have developed the Naccio system for specifying and enforcing security policies. Like SASI, Naccio allows users to add security state to untrusted programs and to define operations that perform security checks. Sandholm and Schwartzbach [23] have developed a system for instrumenting concurrent programs to prevent race conditions and detect violations of safety properties. They take specifications in a second-order modal logic and compile them into a distributed security automaton. Godefroid [4] also verifies properties of concurrent systems. His tool, VeriSoft, detects violations of safety properties using an extended model-checking technique. Finally, Fickas and Feather [3] have developed software that monitors the way a program interacts with its environment. When they detect that the program is not fulfilling its requirements, the information they have collected can be used to help evolve the system. All of these systems ensure some measure of code safety, but none of them actually produce certified code — code that is provably secure and designed to be checked by an independent verifier.

Acknowledgements

I am grateful to Úlfar Erlingsson and Fred Schneider for explaining their security automaton model and SASI system to me. Karl Crary, Úlfar Erlingsson, Neal Glew, Dan Grossman, Greg Morrisett, Stephanie Weirich, Steve Zdancewic, and the anonymous referees have made extremely helpful suggestions on previous drafts of this work.

References

[1] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *ACM International Conference on Functional Programming*, pages 301–312, Baltimore, September 1998.

[2] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, Oakland, May 1999.

[3] Stephen Fickas and Martin Feather. Requirements monitoring in dynamic environments. In *2nd IEEE International Symposium on Requirements Engineering*, March 1995.

[4] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.

[5] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994.

[6] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.

[7] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[8] Dexter Kozen. Efficient code certification. Technical Report TR98-1661, Cornell University, January 1998.

[9] Christopher League, Zhong Shao, and Valery Trifonov. Representing java classes in a typed intermediate language. In *ACM International Conference on Functional Programming*, pages 183–196, Paris, January 1999.

[10] Xavier Leroy. Manifest types, modules, and separate compilation. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 109 – 122, Portland, OR, January 1994.

[11] Xavier Leroy and Francois Rouaix. Security properties of typed applets. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 391–403, San Diego, January 1998.

[12] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[13] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, January 1996.

[14] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Progamming Languages and Systems*, 10(3):470–502, July 1988.

[15] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

[16] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, La Jolla, June 1995.

[17] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998.

[18] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Progamming Languages and Systems*, 21(3):528–569, May 1999.

[19] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[20] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, October 1996.

[21] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM Conference on Programming Language Design and Implementation*, pages 333 – 344, Montreal, June 1998.

[22] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. *LNCS 1419: Special Issue on Mobile Agent Security*, October 1997.

[23] Anders Sandholm and Michael Schwartzbach. Distributed safety controllers for web services. In *Fundamental approaches to Software Engineering*, volume 1382, pages 270–284. Lecture Notes in Computer Science, Springer-Verlag, 1998.

[24] Fred Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, January 1998.

[25] Chris Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, Portland, OR, June 1997.

[26] Frederick Smith, David Walker, and Greg Morrisett. Alias types. Technical Report TR99-1773, Cornell University, October 1999.

[27] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Caledon Hills, September 1999.

[28] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, December 1993.

[29] David Walker. A type system for expressive security policies. Technical Report TR99-1740, Cornell University, April 1999.

[30] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[31] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1999.

[32] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, Quebec, June 1998.

[33] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, January 1999.