JFP **13** (3): 573–600, May 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796802004586 Printed in the United Kingdom

Program specialization for execution monitoring

PETER THIEMANN
Institut für Informatik, Universität Freiburg,
Georges-Köhler-Allee 079, D-79110 Freiburg i.Br., Germany
(*e-mail:* thiemann@informatik.uni-freiburg.de)

Abstract

Execution monitoring is a proven tool for securing program execution and to enforce safety properties on applets and mobile code, in particular. Inlining monitoring tools perform their task by inserting certain run-time checks into the monitored application before executing it. For efficiency reasons, they attempt to insert as few checks as possible using techniques ranging from simple *ad hoc* optimizations to theorem proving. Partial evaluation is a powerful tool for specifying and implementing program transformations. The present work demonstrates that standard partial evaluation techniques are sufficient to transform an interpreter equipped with monitoring code into a non-standard compiler. This compiler generates application code, which contains the inlined monitoring code. If the monitor is enforcing a security policy, then the result is a secured application code. If the policy is defined using a security automaton, then the transformation can elide many run-time checks by using abstract interpretation. Our approach relies on proper staging of the monitoring interpreter. The transformation runs in linear time, produces code linear in the size of the original program, and is guaranteed not to duplicate incoming code.

1 Introduction

Execution monitoring is a well-known methodology for detecting anomalities in running systems (Plattner & Nievergelt, 1981). It is particularly common in operating systems, as well as in the context of distributed systems where it can be hard or even impossible to establish guarantees statically. Recently, it has been applied to enforce security policies on untrusted code fragments received over a network (Erlingsson & Schneider, 1999, 2000).

There are plenty of examples for security policies. Clearly, programs should not violate the access control or other security policies of the computers on which they run, e.g. accessing data or resources in an inappropriate manner.

A promising avenue is to concentrate on security policies that can be enforced by having a reference monitor execute a security automaton (Schneider, 2000). A reference monitor tracks the execution and checks each action for compliance with the security policy before it is performed. If the check fails, the reference monitor terminates the execution. Otherwise, the action is performed and execution continues. This kind of execution monitoring can only enforce *safety properties*, which state that nothing bad will ever happen. Liveness properties, like fair scheduling of resources, are not in the scope of this method[1].

One particular implementation of a reference monitor is code instrumentation (Wahbe *et al.*, 1993; Lee *et al.*, 1999) or inlined reference monitors (Erlingsson & Schneider, 2000). In this approach, before an application is executed, it is first transformed by inserting instructions that perform the monitoring. In all implementations that we know of, this instrumentation is implemented in an *ad hoc* fashion.

We propose a structured approach to inlined reference monitoring. First, we add the reference monitor to an (existing) interpreter, which is easy to do and also easy to examine for correctness. Then, we perform the transformation step automatically using program specialization (partial evaluation). By automating this step, we can be sure that the resulting specialized code

- retains the semantics of all admissible executions (the semantics is not generally preserved because the reference monitor may terminate executions that do not comply with the security policy);
- performs monitoring as implemented in the interpreter; and
- keeps monitoring data and application data apart, so it prevents the subversion or circumvention of the reference monitor.

In the particular case where monitoring implements a security automaton, the partial evaluator can perform some of the compliance checks at transformation time (compile time).

The main goal of the paper is to present this partial-evaluation based approach in a manner as simple as possible. Hence, we rely only on standard features of partial evaluation and abstract interpretation.

Simplicity is also required due to another important concern in the application of monitoring to security: the size of the Trusted Computing Base (TCB) should be as small as possible to enable establishing its correctness by formal verification or by other means. For that reason, many approaches start at the level of (virtual) machine code to avoid having a compiler in the TCB (Erlingsson & Schneider, 1999; Appel & Felty, 2000). Although we are presenting our work in the context of partial evaluation for an applied lambda calculus, our actual implementation relies only on a handful of combinators that implement program specialization and that are proven correct (Thiemann, 1999b). In fact, correct execution of the combinators and of the code generated by them is the only requirement for our approach to work.

- No program analysis (e.g. binding-time analysis to analyze the staging properties of the interpreter) is part of the TCB. The present paper proves all necessary staging properties.
- The approach to program generation scales down to machine code, as was shown by Lee & Leone (1996).
- We are not relying on interprocedural flow analysis (and hence may miss some optimization opportunities) to keep the TCB small and to guarantee linear time execution.

[1] However, some liveness properties may be conservatively approximated.

1.1 Related work

Related work falls in three broad categories. First, we discuss dynamic mechanisms that enforce security policies by performing run-time checks. Secondly, we discuss static mechanisms that trade run-time checks for static analysis. Thirdly, we discuss works related to our implementation vehicle, partial evaluation.

1.1.1 Dynamic approaches

Reference monitors have been conceived and implemented with varying degrees of sophistication. The naive approach to execution monitoring is to run the code on an interpreter that checks each action before actually performing it. This approach is highly flexible but it involves a substantial overhead. Hence, it has never been the design principle of choice for security mechanisms. Our present work shows how this approach can be made practical.

A more common approach is the one taken by the JDK (J2SE, 2000). It equips strategic functions in the library with calls to a security manager. A user-provided instantiation of the security manager is then responsible to keep track of the actions and to stop the code, if necessary. This approach is less flexible, but more efficient. Unfortunately, it requires arguing that there are sufficient calls to the security manager and that they cannot be subverted. Java solves the problem of data and memory integrity statically by subjecting all programs to a bytecode verification process (Lindholm & Yellin, 1996).

The Omniware approach (Wahbe *et al.*, 1993; Adl-Tabatabai *et al.*, 1996; Lucco *et al.*, 1995) guarantees memory integrity by imposing a simple program transformation on programs in assembly language. The transformation confines a foreign module to its own private data and code segment. It could be expressed by partially evaluating a machine-language interpreter with memory-safety checks. The approach is very efficient, but of limited expressiveness. Compared to our work, Omniware works on a lower level (virtual machine language) and implements the transformation in an *ad hoc* way as part of the TCB.

Schneider (2000) considers the kind of security policies that can be enforced using execution monitoring. He shows that only safety properties can be decided and offers a mechanism, security automata, for keeping track of the execution history. His work is inspired by earlier work with Alpern (Alpern & Schneider, 1987). The SASI project implemented this idea (Erlingsson & Schneider, 1999) for x86-assembly language and for JVM bytecode. Both allow for a separate specification of a state automaton and rely on code transformation to integrate the propagation of the state with the execution of the program. They demonstrate their use of partial evaluation to optimize the occurrence of run-time checks with an example, but no further information is given. The authors refined their approach for Java by showing that Java stack inspection can also be enforced efficiently and flexibly by transforming JVM bytecode (Erlingsson & Schneider, 2000). Compared to our work, they allow the specification of a security policy in a separate specification language and they are not specific about the partial evaluation techniques used in their implementation.

In contrast, our security policy is coded into the interpreter[2] and we are explicit about our use of partial evaluation. We take advantage of the latter to construct our formal proofs.

Evans & Twyman (1999) have constructed a system that takes a specification of a safety policy and generates a transformed version of the Java run-time classes. Their safety policies apply on the level of method calls. Any program that uses the transformed classes is guaranteed to obey the specified safety policy. In comparison to our work, their granularity of checking is method calls and there is no formal reasoning about their implementation. In addition, the entire transformer is part of the TCB.

Colcombet & Fradet (2000) propose a framework for enforcing trace properties. Essentially, they transform code to incorporate an inlined reference monitor. The monitor maintains a state which is stepped and checked before executing operations that are relevant to the particular trace property. They describe an automata-based framework for straight-line programs that allows them to express optimizations to the placement of run-time checks in the transformed code. In addition, they extend the framework by considering interprocedural flow. Our approach has similar features and similar expressiveness as their straight-line approach. However, we rely on automatic partial evaluation techniques to achieve many of their optimizations. In addition, we can generate non-standard securing compilers using partial evaluation, whereas their approach is essentially hand-coded.

1.1.2 Static approaches

Many approaches trade run-time checks with static analysis at compile-time. To avoid the problem of increasing the size of the TCB, the static checks yield code annotations that can be quickly checked by a small program before starting the application. No run-time checks need to be executed during actual execution and only the small checker is part of the TCB. However, most work does not consider properties beyond type safety and memory integrity.

Necula and Lee (Necula & Lee, 1998; Necula, 1997) have developed a framework in which compiled machine programs can be combined with an encoding of a proof that the program obeys certain properties (for example, a security policy). Before executing the resulting *proof-carrying code*, the proof must be locally checked against the code. Only this proof checker must be verified and trusted, to make sure that the proof-carrying code obeys the security policy. This has been pursued further down to the semantics of single machine instructions by Appel and others (Michael & Appel, 2000; Appel & Felty, 2000). To date, these approaches have been used to enforce type safety and memory integrity. In contrast, our approach is geared towards monitoring primitive operations and it does not rely on annotations, but rather performs a simple abstract interpretation while generating code.

Kozen (1999) has developed a very light-weight version of proof-carrying code.

[2] It would not be hard to parameterize the interpreter with respect to the security policy, giving rise to a separate specification.

He has built a compiler that includes hints to the structure of the compiled program in the code. A receiver of such instrumented code can verify the structural hints and thus obtain confidence that the program preserves memory integrity.

Typed Assembly Language (TAL) (Morrisett *et al.*, 1998) provides another avenue to generating high-level invariants for low-level code. Using TAL can guarantee type safety and memory integrity. TAL programs include extensive type annotations that enable the receiver to perform type checking effectively.

The capability calculus (Crary *et al.*, 1999) extends typed lambda calculus by an abstract notion of capabilities that are threaded through a computation. The motivating example is memory management, but other capabilities (i.e. states of a security automaton) could be considered. However, more machinery is required for dealing properly with security policies as Walker's work below demonstrates.

Walker (2000) presents a dependent type system that encodes Schneider's security automata in the type-level. Next he shows that the naive monitoring translation, which inserts run-time checks before every primitive operation, yields typable code. He gives a number of examples how the type system can verify the correctness of transformations that remove run-time checks. Most of these transformations require external lemmas about the security policy. These must be proved separately and fed into the system to enable transformations. The type system does not provide guidance as to what transformations are possible. In contrast, our system performs such transformations automatically on straight-line code during the process of inlining the reference monitor. Optimizations that work across procedure boundaries are outside the scope of our approach but can be proved correct in Walker's calculus. On the other hand, a design goal of our transformation is to run in linear time which excludes any but the simplest kind of static analysis, and this also ties well with the simplicity required for a trusted secure system.

1.1.3 Partial evaluation

Implementing program transformations by program specialization has been proposed by Turchin and Glück (Turchin, 1993; Glück, 1994) and put into practice by Glück, Jørgensen, and others (Glück & Jørgensen, 1994b; Glück & Jørgensen, 1994a).

Kishon *et al.* (1991) have investigated the use of partial evaluation for execution monitoring. The objective of their work is to transform an interpreter and a specification for monitoring into a monitoring interpreter. Thus, their work may be regarded as a pre-pass to our work: we start with a particular monitoring interpreter and specialize it to a program with inlined monitoring instructions.

There are some works indicating that partial evaluation can benefit from abstract interpretation (Jones, 1997; Consel & Khoo, 1991). The present work is another evidence that their interplay can be fruitful.

Thiemann (2001) has investigated the use of type specialization to remove (in some cases) all run-time safety checks from programs transformed for execution monitoring. This approach relies on a translation to continuation-passing style that exposes information to the specializer as much as possible. The approach is appealing and works well in many cases, but it is prone to code duplication. Also, due to the sophisticated specializer, the transformation phase may take exponential time, which is not practical.

In contrast, the techniques in the present work are guaranteed to never duplicate monitored code and to perform the transformation in linear time. This is due to the reliance on a less powerful but more efficient specialization technique, which does not perform interprocedural analysis. For example, the technique based on type specialization treats non-recursive functions like straight-line code, whereas the present approach forgets all knowledge about the state of the security automaton across a function call.

1.2 Overview

The present work demonstrates that standard partial evaluation techniques can implement inlined reference monitors by specialization of an interpreter instrumented for execution monitoring. This is advantageous because it avoids ad-hoc solutions by reusing partial evaluation practice (like efficient code generation) and theory. For example, the correctness of a transformation can be shown using standard results from partial evaluation. This allows for modular correctness proofs of the trusted computing base and enables the user of execution monitoring to concentrate on the specification of the security policy and on its correctness.

We start from a naive translation (in section 3) and modify it in section 4 to yield two-level terms (in the sense of the partial evaluation textbook (Jones *et al.*, 1993)), which explicitly manage the computation of a security state. Traditional partial evaluation of these terms implements a translation that never duplicates source code. It yields satisfactory compiled secured programs. We consider a variant of the translation that is more general and yields better results (section 4.2), comparable to the results of Colcombet and Fradet's method for straight-line code (Colcombet & Fradet, 2000).

The purpose of this paper is to demonstrate the concepts involved in a manner as simple as possible. Elsewhere (Thiemann, 2001), we demonstrate the use of an advanced specialization method with a more complicated translation.

The main technical results are the correctness proofs of the translations and the non-standard compilers generated by partial evaluation. They guarantee the safety of the translated code and of the compiled code. In addition, we prove informally that the translation takes $O(n \cdot s^2)$ time where n is the size of the input expression and s is the number of states of the automaton used to define the security policy. We further prove that the translated term has size $O(n)$.

2 Prerequisites

This section introduces the source language, the concept of a security automaton to encode safety properties, and traditional partial evaluation.

Syntax

$$\begin{array}{lccl}
\text{expressions } (\mathit{Exp}) & e & ::= & v \mid (\mathtt{if}\ \ e\ \ e\ \ e) \mid \mathtt{O}(e) \mid e@e \mid (e,\ldots,e) \\
\text{values } (\mathit{Value}) & v & ::= & x \mid a \mid \mathtt{fix}\ f(x,\ldots,x)e \mid (v,\ldots,v) \mid \mathtt{halt} \\
\text{evaluation contexts} & C & ::= & (\mathtt{if}\ \ [\,]\ \ e\ \ e) \mid \mathtt{O}([\,]) \mid [\,]@e \mid v@[\,] \mid \\
 & & & (v,\ldots,[\,],e,\ldots) \\
\text{security states} & \sigma & \in & \mathit{State} \\
\text{base-type constants} & a & \in & \mathit{Base} \\
\text{primitive operators} & \mathtt{O} & \in & \mathit{Op} \\
\text{types} & \tau & ::= & \mathit{BaseType} \mid \tau \to \tau \mid (\tau,\ldots,\tau) \\
\text{traces} & t & ::= & \varepsilon \mid \mathtt{O}(a,\ldots,a) \mid t\,t
\end{array}$$

Labeled reduction rules

$$\begin{array}{lcll}
(\mathtt{if}\ \ \mathtt{true}\ \ e_1\ \ e_2) & \xrightarrow{\varepsilon} & e_1 & \\
(\mathtt{if}\ \ \mathtt{false}\ \ e_1\ \ e_2) & \xrightarrow{\varepsilon} & e_2 & \\
\mathtt{O}(a_1 \ldots a_n) & \xrightarrow{\mathtt{O}(a_1 \ldots a_n)} & a & \text{if } a = [\![\mathtt{O}]\!](a_1,\ldots,a_n) \text{ is defined}
\end{array}$$

$$(\mathtt{fix}\ x_0(x_1,\ldots,x_n)e)@(v_1,\ldots,v_n) \xrightarrow{\varepsilon} e[x_0 \mapsto \mathtt{fix}\ x_0(x_1,\ldots,x_n)e, x_1 \mapsto v_1,\ldots,x_n \mapsto v_n]$$

$$\begin{array}{lcl}
(\mathtt{if}\ \ \mathtt{halt}\ \ e_1\ \ e_2) & \xrightarrow{\varepsilon} & \mathtt{halt} \\
(v_1,\ldots\mathtt{halt}\ldots e_n) & \xrightarrow{\varepsilon} & \mathtt{halt} \\
\mathtt{O}(\mathtt{halt}) & \xrightarrow{\varepsilon} & \mathtt{halt} \\
\mathtt{halt}@e & \xrightarrow{\varepsilon} & \mathtt{halt} \\
v@\mathtt{halt} & \xrightarrow{\varepsilon} & \mathtt{halt}
\end{array}$$

Contextual rules

$$\frac{e \xrightarrow{t} e'}{C[e] \xrightarrow{t} C[e']} \qquad e \xrightarrow{\varepsilon}{}^{*}\ e \qquad \frac{e \xrightarrow{t} e' \quad e' \xrightarrow{t'}{}^{*}\ e''}{e \xrightarrow{tt'}{}^{*}\ e''}$$

Fig. 1. The source language.

2.1 Source language

The source language is a simply-typed call-by-value lambda calculus with constants, conditionals, tuples, and primitive operations on tuples of base-type values (see figure 1). The expression $\mathtt{fix}\ x_0(x_1,\ldots,x_n)e$ denotes a recursively defined function. We write $\lambda(x_1,\ldots,x_n)e$ if x_0 does not appear in e, and $\mathtt{let}\ (x_1,\ldots,x_n) = e_1\ \mathtt{in}\ e_2$ for $(\lambda(x_1,\ldots,x_n)e_2)@e_1$. The latter is the only way to inspect the components of a tuple. The standard typing rules defining the judgement $\Gamma \vdash e : \tau$ are omitted. The only exception is the exceptional value `halt` which stops the program when it appears in an evaluation context. Since its value is never consumed, `halt` can assume any type.

$$\Gamma \vdash \mathtt{halt} : \tau$$

The semantics is defined by a labeled transition relation $\xrightarrow{t}$, where each label, t, is a sequence of primitive operations and their arguments. The label, ε, denotes the empty sequence. Every primitive operation, $\mathtt{O} : \mathit{BaseType}^n \rightarrow \mathit{BaseType}$, is defined through an associated partial semantic function $[\![\mathtt{O}]\!] \in \mathit{BaseType}^n \hookrightarrow \mathit{BaseType}$. The reduction rule for recursive function application relies on simultaneous, capture-avoiding substitution $e[x_0 \mapsto \mathtt{fix}\ x_0(x_1, \ldots, x_n)e, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$. The first four rules are the usual execution rules for conditionals, primitive operations, and recursive functions with call-by-value semantics. The second group of rules specifies the propagation of `halt`. It behaves like an exception: it pops its entire evaluation context until only `halt` remains at last.

Each reduction sequence $e_0 \xrightarrow{t_1} e_1 \xrightarrow{t_2} \ldots$ gives rise to a potentially infinite sequence $\vec{t} = (t_1, t_2, \ldots)$ of primitive operations (an execution *trace*). We write $e \downarrow^t v$ if there is a finite sequence of reductions, $e \xrightarrow{t}{}^{*}\ v$.

2.2 Security automata

According to Schneider (2000), a *security policy* $\mathscr{P}$ maps a set, Π, of finite and infinite execution traces to a truth value (*true* or *false*). We say that Π satisfies $\mathscr{P}$ if $\mathscr{P}(\Pi) = \mathit{true}$. A policy $\mathscr{P}$ is a *property* if satisfaction is defined element-wise (Alpern & Schneider, 1985). That is, there is a function $\widehat{\mathscr{P}}$ mapping an execution trace, σ, to a truth value so that $\mathscr{P}(\Pi) = (\forall \sigma \in \Pi)\ \widehat{\mathscr{P}}(\sigma)$. Hence, a reduction sequence is acceptable with respect to a property if $\widehat{\mathscr{P}}$ of its trace yields *true*. *Safety properties* are particular properties which place further restrictions on the set of traces (Schneider, 2000). Schneider has further shown that execution monitoring can only enforce safety properties. He proceeds to demonstrate that execution monitoring can be modeled using a *security automaton*, *i.e.*, a state automaton with a distinguished sink state *bad*. The sink state models failure to comply with the security policy.

Following Walker (2000), a security automaton is a tuple $\mathscr{S} = (\mathit{State}, \mathit{Op}, \mathit{Value}, \delta, \sigma_0, \mathit{bad})$, where

- *State* is a countable set of states;
- *Op* is a finite set of operation symbols;
- *Value* is a countable set of values;
- $\delta : \mathit{Op} \times \mathit{Value}^* \times \mathit{State} \rightarrow \mathit{State}$ is a total function so that $(\forall \mathtt{O} \in \mathit{Op}, x_1 \ldots x_n \in \mathit{Value}^*)\ \delta(\mathtt{O}, x_1 \ldots x_n, \mathit{bad}) = \mathit{bad}$ (state transition function);
- $\sigma_0 \in \mathit{State}$ is the initial state; and
- $\mathit{bad} \in \mathit{State}$ is the sink state with $\sigma_0 \neq \mathit{bad}$.

For a trace, t, define the iterated transition function $\delta^*(t, \sigma)$ by

$$\begin{array}{lcl} \delta^*(\varepsilon, \sigma) & = & \sigma \\ \delta^*(\mathtt{O}(a_1 \ldots a_n), \sigma) & = & \delta(\mathtt{O}, a_1 \ldots a_n, \sigma) \\ \delta^*(t_1 t_2, \sigma) & = & \delta^*(t_2, \delta^*(t_1, \sigma)) \end{array}$$

A closed term e is *safe* with respect to $\mathscr{S}$ and some $\sigma \in \mathit{State} \setminus \{\mathit{bad}\}$ if for all traces t and expressions e' where $e \xrightarrow{t}{}^{*}\ e'$, it holds that $\delta^*(t, \sigma) \neq \mathit{bad}$. It is safe with respect to $\mathscr{S}$ if it is safe with respect to the initial state σ_0.

A typical example (Schneider, 2000; Erlingsson & Schneider, 1999; Walker, 2000) is the policy that no network `send` operation happens after a `read` operation from a local file.

The set of states has three elements

$$State = \{before\text{-}read, after\text{-}read, bad\}$$

with initial state $\sigma_0 = before\text{-}read$. The transition functions are the identity functions for all primitive operations except `send` and `read`:

σ	$\delta(\texttt{read}, file, \sigma)$	$\delta(\texttt{send}, data, \sigma)$	$\delta(\texttt{O}, y_1 \dots y_n, \sigma)$
before-read	*after-read*	*before-read*	*before-read*
after-read	*after-read*	*bad*	*after-read*
bad	*bad*	*bad*	*bad*

The program $(\lambda(x)\texttt{read}(file))@(\texttt{send}(data))$ is safe (with respect to σ_0) since its trace is $t = (\texttt{send}(data), \texttt{read}(file))$ and $\delta^*(t, \sigma_0) = after\text{-}read$. It is not safe with respect to *after-read* since $\delta^*(t, after\text{-}read) = bad$.

The program $(\lambda(x)\texttt{send}(data))@(\texttt{read}(file))$ is not safe with respect to any state: its trace $t' = (\texttt{read}(file), \texttt{send}(data))$ yields $\delta^*(t', before\text{-}read) = bad$ as well as $\delta^*(t', after\text{-}read) = bad$.

2.3 Partial evaluation

Traditional partial evaluation techniques (Consel & Danvy, 1993; Jones *et al.*, 1993) rely on non-standard interpretation of a source program to perform as many operations on compile-time data as possible. In *offline partial evaluation*, a binding-time analysis determines the compile-time operations before actually specializing the code. The analysis communicates with the specializer using two-level terms, which indicate code generation by underlining and compile-time operations by overlining. Interpretation of an underlined term first interprets the subterms and then constructs a syntax tree from the values according to the underlined term constructor. Interpretation of an overlined term is identical to standard interpretation.

Although offline partial evaluation misses some specialization opportunities, it demonstrates the concepts involved in compiling monitoring execution well:

- It is simple to understand as non-standard interpretation.
- It is easy to predict the form and the size of the compiled code as well as the compilation speed by inspecting the two-level term.
- It gives satisfactory results for this application.
- It is possible to circumvent the binding-time analysis and to directly generate two-level terms if it can be proven in advance that they are well-annotated with respect to the analysis.
- Specialization is very fast.

We rely on the PGG system (Thiemann, 2000), which specializes quickly because it relies on the *cogen approach* to program specialization (Launchbury & Holst, 1991). In this approach, there is no interpretation of two-level terms. Instead, two-level terms are executed directly using a fast implementation of the code generating constructs (Thiemann, 1999b).

Proving the correctness of a partial evaluator amounts to proving the MIX-equation (Jones *et al.*, 1993). Suppose that `spec` is the program text of a specializer and p is a two-level term denoting a program with inputs $\vec{x}$ and $\vec{y}$. Let further $[\![\cdot]\!]$ map a program text to a function from inputs to output. Considering the inputs $\vec{x}$ as compile-time inputs and the inputs $\vec{y}$ as run-time inputs, the MIX-equation reads

$$[\![[\![\texttt{spec}]\!]\ p\ \vec{x}]\!]\ \vec{y} = [\![\mathit{erase}(p)]\!]\ \vec{x}\ \vec{y}. \tag{1}$$

The function *erase*() maps a two-level term to a standard term by erasing all overlining and underlining annotations as well as $\mathit{erase}(\texttt{lift}\ e) = \mathit{erase}(e)$ (The construct `lift` y converts a compile-time value, y, into a literal with the same run-time value.). In reading this equation, it is assumed that the specializer is terminating, i.e. $p' = [\![\texttt{spec}]\!]\ p\ \vec{x}$ always yields a result. Furthermore, the specialized program, p', is assumed to terminate on input $\vec{y}$ if and only if the original program, p, terminates on input $\vec{x}\ \vec{y}$.

The text of the specialized (compiled) program is $[\![\texttt{spec}]\!]\ p\ \vec{x}$. The cogen approach mentioned above transforms a two-level program p into another program g, which is equivalent to $[\![\texttt{spec}]\!]\ p$, but is 6–8 times faster (Thiemann, 1999b). Hence, g takes the compile-time inputs $\vec{x}$ and produces the specialized program. The program g is called a *generating extension* for p.

The correctness of offline partial evaluation has been considered in a number of places (Hatcliff & Danvy, 1997; Hatcliff, 1995; Consel & Khoo, 1995; Lawall & Thiemann, 1997). In this work, we take the correctness of the underlying specializer for granted and make use of (1) where required. Since this might raise concerns that the specializer becomes part of the TCB, we remark that (i) specializers have been mechanically proves correct (Hatcliff, 1995) and (ii) by relying on the cogen approach mentioned above, it is only necessary to verify the building bricks of the generating extension, as we demonstrated elsewhere (Thiemann, 1999b).

3 Execution monitoring

One way to enforce safe execution is to install a reference monitor that observes the execution and halts the system whenever it is about to violate a security policy. In an interpreted setting based on security automata, the reference monitor can be included in the interpreter. In the terminology of Erlingsson & Schneider (2000),

- the *security events* are the executions of primitive operations,
- the *security state* is the state of the security automaton, and
- the *security updates* are the extra constructs in the handling of primitive operations that maintain the state of the security automaton: Before attempting a primitive operation, the interpreter steps the security state according to the operation and its arguments and checks whether the result is *bad*. If that is the

case, the program is stopped. Otherwise, the interpreter performs the primitive operation and continues using the new security state.

The interpreted setting seems well-suited to implementing such a reference monitor because

1. the interpreter clearly mediates all security events;
2. the integrity of the interpreter is protected from malevolent applications; and
3. the presence of the interpreter is transparent to the application.

Erlingsson & Schneider (2000) continue to consider Inlined Reference Monitors (IRMs). We are also aiming at providing an IRM, but instead of implementing a system to instrument code with an IRM from scratch, we obtain such a system by specializing an interpreter that includes a reference monitor. Another view of this specialization process is to consider an interpreter and the translation achieved by unfolding the interpreter's defining equations interchangeably. This view is quite natural because partial evaluation of an interpreter implements exactly this translation.

3.1 Naive translation

Figure 2 shows a translation that makes the sequencing of the operations and the passing of the security state explicit. The output of the translation is in A-normal form (Flanagan *et al.*, 1993), and thus well suited to compiling it. In particular, the target language of the translation is a typed lambda calculus (like the source language) extended by a separate `let` expression with the obvious derived evaluation rule. A `let` expression is compiled to more efficient code than a function application.

The translation is specified using two pairs of mutually recursive functions, $\|\cdot\|$ and $|\cdot|$. One pair of functions works on the type level and the other on the expression level. In general, $\|\cdot\|$ translates values and types for values, whereas $|\cdot|$ translates computations and their types. They are inspired by monadic translations (Hatcliff & Danvy, 1994). The value translation for a type leaves most parts unchanged, except for function types where the argument type is translated using $\|\cdot\|$ and the result type is translated using $|\cdot|$.

The translation of a computation yields a (security) state transformer. It is a function that accepts a state and returns the updated state paired with a value of suitable type.

Variables only contain values. Hence, all type assumptions on variables are translated using $\|\cdot\|$.

The translation of value expressions is straightforward. Variables and base type constants are passed through unchanged and, for a tuple, the translation rebuilds the tuple with the translated values. Recursive functions are mapped into recursive functions that take an extra parameter, the security state σ, and have their body translated using the translation of computation expressions, as required by the type translation outlined above.

The translation of computation expressions takes a security state as a parameter

Types

$$\begin{array}{lcl}
||\mathit{BaseType}|| & = & \mathit{BaseType} \\
||\tau_1 \to \tau_2|| & = & ||\tau_1|| \to |\tau_2| \\
||(\tau_1, \dots, \tau_n)|| & = & (||\tau_1||, \dots, ||\tau_n||) \\
|\tau| & = & \mathit{State} \to (\mathit{State}, ||\tau||)
\end{array}$$

Type environments

$$\begin{array}{lcl}
||\emptyset|| & = & \emptyset \\
||\Gamma, x : \tau|| & = & ||\Gamma||, x : ||\tau||
\end{array}$$

Values

$$\begin{array}{lcl}
||x|| & = & x \\
||a|| & = & a \\
||\texttt{fix}\ x_0(x_1 \dots x_n)e|| & = & \texttt{fix}\ x_0(x_1, \dots, x_n)\lambda(\sigma)|e|\sigma \\
||(v_1, \dots, v_n)|| & = & (||v_1||, \dots, ||v_n||)
\end{array}$$

Terms

$$\begin{array}{lcl}
|v|\sigma & = & (\sigma, ||v||) \\
|(\texttt{if}\ e_1\ e_2\ e_3)|\sigma & = & \texttt{let}\ (\sigma_1, y_1) = |e_1|\sigma\ \texttt{in} \\
 & & \texttt{if}\ y_1\ \texttt{then}\ |e_2|\sigma_1\ \texttt{else}\ |e_3|\sigma_1 \\
|(e_1, \dots, e_n)|\sigma & = & \texttt{let}\ (\sigma_1, y_1) = |e_1|\sigma\ \texttt{in}\ \dots \\
 & & \texttt{let}\ (\sigma_n, y_n) = |e_n|\sigma_{n-1}\ \texttt{in} \\
 & & (\sigma_n, (y_1, \dots, y_n)) \\
|\texttt{O}(e)|\sigma & = & \texttt{let}\ (\sigma', y) = |e|\sigma\ \texttt{in} \\
 & & \texttt{let}\ \sigma'' = \delta(\texttt{O}, y, \sigma')\ \texttt{in} \\
 & & \texttt{if}\ \sigma'' = \mathit{bad}\ \texttt{then halt else}\ (\sigma'', \texttt{O}(y)) \\
|e_0 @ e_1|\sigma & = & \texttt{let}\ (\sigma_0, y_0) = |e_0|\sigma\ \texttt{in} \\
 & & \texttt{let}\ (\sigma_1, y_1) = |e_1|\sigma_0\ \texttt{in} \\
 & & y_0 @ y_1 @ \sigma_1
\end{array}$$

Fig. 2. Monitoring interpreter.

and constructs a pair containing the updated state and the actual result. The cases for values, conditional, tuples, and application are standard, i.e. as usual in a monadic translation to state-passing style. The case for primitive operations inserts the stepping and testing of the security state as explained above: It evaluates the argument to the primitive, determines the security state *after* execution of the primitive, and stops execution if this results in a *bad* state. Otherwise, it continues with the new security state and the result of the operation.

3.2 *Properties of the naive translation*

The translation acts on types as follows.

Proposition 3.1

1. If $\Gamma \vdash e : \tau$ then $||\Gamma|| \vdash |e| : |\tau|$.
2. If $\Gamma \vdash v : \tau$ then $||\Gamma|| \vdash ||v|| : ||\tau||$.

A translated program is always safe with respect to the underlying security automaton because it has reified the security state and explicitly checks this state before attempting an operation. The check itself does not subvert safety because the primitive operation δ on the explicit security state does not affect the implicit state of the security automaton. Formally, let $\mathcal{S}' = (State, Op', Value, \delta', \sigma_0, bad)$ with $Op' = Op \cup \{\delta\}$ where δ' is specified by $\delta'(\mathtt{O}, y_1 \dots y_n, \sigma) = \delta(\mathtt{O}, y_1 \dots y_n, \sigma)$, for all $\mathtt{O} \in Op$, and $\delta'(\delta, y_0 y_1 \dots y_n, \sigma) = \sigma$. While the automaton $\mathcal{S}$ determines the safety of the original expression, the automaton $\mathcal{S}'$ determines the safety of the translated expression.

We conclude by stating the key properties of the translation. An expression translated with state σ is safe with respect to $\mathcal{S}'$ and σ.

Proposition 3.2
Let $\sigma \neq bad$. If $|e|\sigma \downarrow^{t'} (\sigma', v')$ then $\delta'^*(t', \sigma) = \sigma'$ where $\sigma' \neq bad$.

Proof
By induction on the number of steps needed for $|e|\sigma \downarrow^{t'} (\sigma', v')$ and a case analysis on e. $\square$

For the following propositions, we need a simple lemma. It states that substitution of values is compatible with the translation.

Lemma 3.3

1. $|e|\sigma[x := ||v'||] = |e[x := v']|\sigma$
2. $||v||\sigma[x := ||v'||] = ||v[x := v']||\sigma$

If the original term evaluates to a value without entering a bad state then so does the translated term. For first-order results, the values of both computations are equal. For functional results, the values are related by $||\cdot||$. To state this connection precisely, it is necessary to relate execution traces for $\mathcal{S}$ with those of $\mathcal{S}'$. If t is a trace for $\mathcal{S}$ and σ is a security state then define $|t|\sigma$ as follows:

$$\begin{array}{lcl} |\varepsilon|\sigma & = & \varepsilon \\ |\mathtt{O}(a_1 \dots a_n)|\sigma & = & \delta(\mathtt{O}, a_1 \dots a_n, \sigma)\ \mathtt{O}(a_1 \dots a_n) \\ |t_1 t_2|\sigma & = & |t_1|\sigma\ |t_2|(\delta^*(t_1, \sigma)) \end{array}$$

Hence, each operation in a trace is transformed into its checking operation $\delta(\mathtt{O}, \dots)$ followed by the actual operation. First, we need a lemma that shows that the translated term simulates the behavior of the original term in some way.

Lemma 3.4
Suppose that $e \xrightarrow{t} e'$ and $\sigma' = \delta(t, \sigma) \neq bad$.

Then $|e|\sigma \xrightarrow{|t|\sigma}{}^{*} |e'|\sigma'$.

Proof
By induction on the definition of $\xrightarrow{t}$. $\square$

A further induction on the definition of $\xrightarrow{t}{}^{*}$ yields the corresponding result for values.

Proposition 3.5
Let $\sigma \neq \mathit{bad}$. Suppose $e \downarrow^{t} v$ and $\sigma' = \delta^{*}(t,\sigma)$.
If $\sigma' \neq \mathit{bad}$ then $|e|\sigma \downarrow^{t'} (\sigma', ||v||)$ where $t' = |t|\sigma$.

If evaluation of the translation of a typed term leads to non-termination or to an undefined primitive operation then so does evaluation of the source term.

Proposition 3.6
Suppose $\emptyset \vdash e : \tau$. If there exist no σ', v', and t' such that $|e|\sigma \downarrow^{t'} (\sigma', v')$ then there exist no v and t such that $e \downarrow^{t} v$.

4 Compiling policies by partial evaluation

The naive translation (figure 2) is already amenable to partial evaluation, by considering the functions $|\cdot|$ and $||\cdot||$ as compile-time functions and unfolding their invocations. The resulting programs are safe but inefficient because they test the security state at every primitive operation.

To achieve more interesting results, we must transform the naive translation to (1) expose compile-time information about the security state, and to (2) stage all operations to make sure that compile-time operations do not depend on run-time values. The first task requires some creativity and insight in the particular problem. The second task is performed with the help of a type system or a binding-time analysis.

In particular, we must decide which values should become compile-time values. Quite often, the most obvious choice does not yield a correct staging (a violation of (2)) or it leads to an ill-behaved specialization: Either too much is specialized so that code is duplicated or too little is specialized so that only trivial computations are performed at specialization time. In such a case, the two-level program must be augmented to compute additional values at compile-time. Determining these values and taking advantage of them requires creativity and insight into the problem.

The revised translation makes the staging explicit using a two-level language. In a two-level program, overlining indicates compile-time operations whereas underlining indicates run-time operations. The same markup applies to types, too. Running a two-level program executes the compile-time operations and generates code for the run-time operations. Invocations of the translation are considered as compile-time operations. Specializing the translation with respect to an incoming program results in an optimized, secured program.

The translation that we are showing is written in a two-level language, that is, it is already binding-time analyzed. This is proves in Proposition 4.1 in section 4.3. The binding-time analysis that we are using relies on context propagation (or continuation-based partial evaluation (Bondorf, 1992)), a standard feature of state-of-the-art partial evaluators. Inspection of the translation reveals that compile-time values are returned from the bodies of run-time let expressions. Context propagation enables the partial evaluator to float the run-time let expression outward so that the compile-time computation can continue in the body of the let expression. Technically,

Types

$$\begin{array}{lcl}
\|BaseType\|_o & = & \underline{BaseType} \\
\|\tau_1 \rightarrow \tau_2\|_o & = & (\|\tau_1\|_o)\underline{\rightarrow}|\tau_2|_d \\
\|(\tau_1, \ldots, \tau_n)\|_o & = & (\|\tau_1\|_o, \ldots, \|\tau_n\|_o) \\
|\tau|_d & = & (\underline{State})\underline{\rightarrow}(\underline{State}, \|\tau\|_o) \\
|\tau|_o & = & \quad\quad \langle \overline{\mathscr{P}(State)}, \underline{State}\overline{\rightarrow}\underline{State}, \underline{State}\rangle \\
 & & \overline{\rightarrow} \ \ \langle \overline{\mathscr{P}(State)}, \underline{State}\overline{\rightarrow}\underline{State}, \underline{State}, \|\tau\|_o\rangle
\end{array}$$

Type environments

$$\begin{array}{lcl}
\|\emptyset\|_o & = & \emptyset \\
\|\Gamma, x : \tau\|_o & = & \|\Gamma\|_o, x : \|\tau\|_o
\end{array}$$

Fig. 3. Staged translation I: types and environments.

the partial evaluator "collects" the let-definitions in the nearest enclosing run-time expression (usually a run-time function or conditional).

4.1 Staged translation

The first idea is to make the security state a compile-time value. However, this approach quickly leads to problems. In a traditional partial evaluator, run-time data structures cannot contain compile-time values. In particular, run-time functions neither take compile-time parameters nor deliver compile-time results. Since the translation must treat all functions from the incoming code as run-time functions, it is not possible to thread a compile-time state through them (at least not without code duplication, which we want to avoid).

The next idea is to represent the changes to the security state by a compile-time value. Unfortunately, even that is not possible, in general, because δ can depend upon the actual (run-time) arguments to a primitive operation.

However, we can consider the compile-time component, $\overline{\delta}(\mathtt{O}) : \overline{State} \rightarrow \overline{State}$, of the transition function, δ. It is defined by

$$\overline{\delta}(\mathtt{O})(\overline{\sigma}) := \begin{cases} \overline{\sigma}' & \text{if } \forall y_1 \ldots y_n.\delta(\mathtt{O}, y_1, \ldots, y_n, \sigma) = \sigma' \\ bad & \text{otherwise} \end{cases}$$

The first case applies if the next state is independent of the runtime arguments, $y_1, \ldots, y_n$, to $\mathtt{O}$, so it can be computed at compile-time. In the second case, the next state depends upon the runtime arguments so it must be computed at run-time. The latter is signaled by returning *bad* at compile-time.

Building on $\overline{\delta}(\mathtt{O})$, our translation (shown in figures 3 and 4) transforms incoming code using three parameters:

1. A compile-time set, $\overline{S} \in \overline{\mathscr{P}(State)}$, of possible current security states.
2. A compile-time function, $d \in \underline{State}\overline{\rightarrow}\underline{State}$, that maps dynamic security states.
3. A run-time value σ that holds a dynamic security state.

The same three values are also returned (together with the computed value) from the translated two-level expression. The main trick of the translation is to handle

Values

$$
\begin{array}{lcl}
||x||_o & = & x \\
||a||_o & = & a \\
||\mathtt{fix}\ x_0(x_1 \ldots x_n)e||_o & = & \underline{\mathtt{fix}}\ x_0(x_1,\ldots,x_n)\underline{\lambda}\sigma. \\
 & & \quad \overline{\mathtt{let}}\ \langle \overline{S}, d, \sigma', y\rangle = |e|_o \overline{@} \langle \mathit{State} \setminus \{\mathit{bad}\}, \overline{\lambda}\sigma.\sigma, \sigma\rangle \\
 & & \quad \overline{\mathtt{in}}\ (d\overline{@}(\sigma'), y) \\
||(v_1,\ldots,v_n)||_o & = & (||v_1||_o,\ldots,||v_n||_o)
\end{array}
$$

Terms

$$
\begin{array}{lll}
\multicolumn{3}{l}{|v|_o \overline{@} \langle \overline{S}, d, \sigma\rangle} \\
 & = & \langle \overline{S}, d, \sigma, ||v||_o\rangle \\
\multicolumn{3}{l}{|(\mathtt{if}\ e_1\ e_2\ e_3)|_o \overline{@} \langle \overline{S}, d, \sigma\rangle} \\
 & = & \overline{\mathtt{let}}\ \langle \overline{S}_1, d_1, \sigma_1, y_1\rangle = |e_1|_o \overline{@} \langle \overline{S}, d, \sigma\rangle\ \overline{\mathtt{in}} \\
 & & \underline{\mathtt{let}}\ (\sigma', y') = \\
 & & \quad \underline{\mathtt{if}}\ y_1 \\
 & & \quad \underline{\mathtt{then}}\ \overline{\mathtt{let}}\ \langle \overline{S}_2, d_2, \sigma_2, y_2\rangle = |e_2|_o \overline{@} \langle \overline{S}_1, d_1, \sigma_1\rangle\ \overline{\mathtt{in}}\ (d_2\overline{@}(\sigma_2), y_2) \\
 & & \quad \underline{\mathtt{else}}\ \overline{\mathtt{let}}\ \langle \overline{S}_3, d_3, \sigma_3, y_3\rangle = |e_3|_o \overline{@} \langle \overline{S}_1, d_1, \sigma_1\rangle\ \overline{\mathtt{in}}\ (d_3\overline{@}(\sigma_3), y_3) \\
 & & \underline{\mathtt{in}}\ \langle \mathit{State} \setminus \{\mathit{bad}\}, \overline{\lambda}\sigma.\sigma, \sigma', y'\rangle \\
\multicolumn{3}{l}{|(e_1,\ldots,e_n)|_o \overline{@} \langle \overline{S}, d, \sigma\rangle} \\
 & = & \overline{\mathtt{let}}\ \langle \overline{S}_1, d_1, \sigma_1, y_1\rangle = |e_1|_o \overline{@} \langle \overline{S}, d, \sigma\rangle\ \overline{\mathtt{in}}\ \ldots \\
 & & \overline{\mathtt{let}}\ \langle \overline{S}_n, d_n, \sigma_n, y_n\rangle = |e_n|_o \overline{@} \langle \overline{S}_{n-1}, d_{n-1}, \sigma_{n-1}\rangle\ \overline{\mathtt{in}} \\
 & & \langle \overline{S}_n, d_n, \sigma_n, (y_1,\ldots,y_n)\rangle \\
\multicolumn{3}{l}{|\mathtt{O}(e)|_o \overline{@} \langle \overline{S}, d, \sigma\rangle} \\
 & = & \overline{\mathtt{let}}\ \langle \overline{S}', d', \sigma', y\rangle = |e|_o \overline{@} \langle \overline{S}, d, \sigma\rangle\ \overline{\mathtt{in}} \\
 & & \overline{\mathtt{let}}\ \overline{S}'' = \overline{\delta}(\mathtt{O})(\overline{S}')\ \overline{\mathtt{in}} \\
 & & \overline{\mathtt{let}}\ d'' = \overline{\delta}(\mathtt{O}) \overline{\circ} d'\ \overline{\mathtt{in}} \\
 & & \overline{\mathtt{if}}\ \mathit{bad} \notin \overline{S}''\ \overline{\mathtt{then}}\ \underline{\mathtt{let}}\ y' = \underline{\mathtt{O}}(y)\ \underline{\mathtt{in}}\ \langle \overline{S}'', d'', \sigma', y'\rangle\ \overline{\mathtt{else}} \\
 & & \underline{\mathtt{let}}\ \sigma'' = \delta(\mathtt{O}, y, d'\overline{@}(\sigma'))\ \underline{\mathtt{in}} \\
 & & \underline{\mathtt{if}}\ \sigma'' \underline{=} \mathtt{lift}\ \mathit{bad}\ \underline{\mathtt{then}}\ \underline{\mathtt{halt}}\ \underline{\mathtt{else}} \\
 & & \underline{\mathtt{let}}\ y' = \underline{\mathtt{O}}(y)\ \underline{\mathtt{in}} \\
 & & \langle \overline{S}'' \setminus \{\mathit{bad}\}, \overline{\lambda}\sigma.\sigma, \sigma'', y'\rangle \\
\multicolumn{3}{l}{|e_0 @ e_1|_o \overline{@} \langle \overline{S}, d, \sigma\rangle} \\
 & = & \overline{\mathtt{let}}\ \langle \overline{S}_0, d_0, \sigma_0, y_0\rangle = |e_0|_o \overline{@} \langle \overline{S}, d, \sigma\rangle\ \overline{\mathtt{in}} \\
 & & \overline{\mathtt{let}}\ \langle \overline{S}_1, d_1, \sigma_1, y_1\rangle = |e_1|_o \overline{@} \langle \overline{S}_0, d_0, \sigma_0\rangle\ \overline{\mathtt{in}} \\
 & & \underline{\mathtt{let}}\ (\sigma', y') = y_0 \underline{@} (y_1) \underline{@} (d_1 \overline{@} (\sigma_1))\ \underline{\mathtt{in}} \\
 & & \langle \mathit{State} \setminus \{\mathit{bad}\}, \overline{\lambda}\sigma.\sigma, \sigma', y'\rangle
\end{array}
$$

Fig. 4. Staged translation II: expressions.

updates to σ (which necessarily happen at run-time) lazily. At any point in the execution, σ contains a *past* security state. The current security state is always represented by $d\overline{@}(\sigma)$ and is approximated by $\overline{S}$. The state σ is only current when it is completely unknown, that is, on entry to a function, on return from a function, or after a primitive operation that cannot be checked at compile-time.

For easier readability, we have chosen *not* to use the overlining/underling markup for tuples. Instead, we write $\langle \overline{S}, d, \sigma\rangle$ for constructing and examining compile-time tuples and (σ', y') for run-time tuples.

Although d acts on run-time values, it is a compile-time value. It can be composed and examined at compile-time.

The translated term uses the set $\overline{S}$ at compile-time to predict whether a run-time test for *bad* is necessary. Formally, the translation maintains the following compile-time invariants:

1. the current state is equal to $d\overline{@}\sigma$,
2. the current state is contained in $\overline{S}$: $d\overline{@}\sigma \in \overline{S}$, and
3. $bad \notin \overline{S}$.

To initialize these invariants, the approximation $\overline{S}$ is set to $State \setminus \{bad\}$[3], whenever the current security state is completely unknown. This happens in the translation of `fix`, function application, the conditional (to some extent), and the primitive operations.

To maintain the invariants throughout the translation, each application of a primitive updates $\overline{S}$ by computing[4]

$$\overline{\delta}(\mathtt{O})(\overline{S}) := \bigcup_{\overline{\sigma} \in \overline{S}} \overline{\delta}(\mathtt{O})(\overline{\sigma}).$$

In addition, the translation composes d with the static state transition function $\overline{\delta}$ of the primitive operation. Only if the test for *bad* is not possible at compile-time does the translated term apply d to the dynamic security state σ to compute the current security state at run-time and perform the test.

The translation of values, $\|\cdot\|_o$, does not involve passing of state information. Except for `fix`, the cases are straightforward. In the case of `fix`, each function receives an additional argument and an additional result, both of type $\underline{State}$ (a run-time security state). Since the security state passed to a function is unknown at compile-time, the corresponding argument σ represents the security state at the time the function is called. To transform the body of the function, we start with the set $State \setminus \{bad\}$ and the identity transition function $\overline{\lambda}\sigma.\sigma$ which verifies the invariant that $(\overline{\lambda}\sigma.\sigma)\overline{@}(\sigma) = \sigma$ is the current state. Correspondingly, at the end of the computation in the body, the function can only return run-time values. Hence, the translated term applies the accumulated compile-time transition function to the run-time state to return the current run-time state to the caller. We call this *flushing the compile-time information*. Dually, in the translation of a function application, the current state is passed to the function as a run-time value and the returned state is paired up with a compile-time identity as transition function.

This explains the two different translations for the type of an expression, $|\cdot|_d$ and $|\cdot|_o$. The translation $|\cdot|_d$ applies to the result of a run-time function and thus returns a run-time type, where the compile-time parts have been flushed as just explained. The translation $|\cdot|_o$ yields a two-level type with non-trivial staging, i.e.

[3] It would be possible to do better here by designing an abstract interpretation that computes a conservative estimate of the set of possible states on entry to each body of a function. However, this approach requires an additional prepass involving a fixpoint computation. Thus, it would break the conceptual simplicity of the approach as well as the linear time complexity.

[4] This computation can take time quadratic (or worse) in the size of *State*. To bring this time down to a constant, we can conservatively approximate $\overline{\delta}(\mathtt{O})(\overline{S})$ by $\overline{\delta}(\mathtt{O})(State \setminus \{bad\})$, which can be precomputed and then looked up at translation time.

with compile-time and run-time parts in expressions that do not immediately crop up as the body of a function.

The translation of a value as a computation just passes on the approximation, the state, and the transition function without change.

The translation of a conditional also involves flushing. It threads approximation, state, and translation functions through the computation of the conditional and passes it to one of the branches according to the outcome of the condition. Since each of the branches may compute a different compile-time transition function (remember that the two-level language executes *both* branches of a dynamic conditional), the translation flushes both of them to return just a dynamic state in σ'. This state is then packaged with $State \setminus \{bad\}$ and a compile-time identity to satisfy the invariant once again.

The translation of a tuple that contains non-values threads the compile-time information through the evaluation from left to right.

The translation of a primitive operation is the most complicated part. Suppose the translated term is about to execute a primitive operation, $\mathtt{O}$. Let $\overline{S}'$, d', and σ' be the approximation, the transition function, and the state after evaluating the argument of $\mathtt{O}$. First, the translated term computes $\overline{S}'' = \overline{\delta}(\overline{S}')$, the approximate state *after* applying the primitive. Next, it composes the static transition function for $\mathtt{O}$ with the current transition function: $d'' = \overline{\delta}(\mathtt{O})\overline{\circ}d'$. To test whether $d''(\sigma')$ (the state after the operation) is not *bad*, it is sufficient to test if $bad \notin \overline{S}''$, by the invariant. If this test succeeds, the operation is performed and the compile-time transition is updated to d'' in the rest of the computation. Otherwise, the run-time state is first bumped to $d'(\sigma')$, the current run-time state. Then, the translated term applies the dynamic state transition function for $\mathtt{O}$ to the actual parameters y and the current run-time state. Hence, the run-time state after the operation is $\sigma'' = \delta(\mathtt{O}, y, d'(\sigma'))$. This value is tested for *bad* at run-time and the computation is halted if that is the case. Otherwise, the translated term performs the operation and the compile-time transition is reset to the identity function.

Function application first threads the translation arguments through the computation of the function and the argument. Next, it flushes the state and passes it as a run-time argument to the function. On return from the function, there is no information about σ', so its approximation has to start with $State \setminus \{bad\}$.

The implementation encodes d as a list of primitive operators in reverse order of execution. The meaning of the list $\mathtt{O}_n \dots \mathtt{O}_1$ is the composition of the associated static state transition functions $\overline{\delta}(\mathtt{O}_n)\overline{\circ}\dots\overline{\circ}\overline{\delta}(\mathtt{O}_1)$ and the empty list stands for $\overline{\lambda}\sigma.\sigma$. This representation enables simple computation of $d(\sigma)$ regardless of σ's binding time and it is amenable to memoization.

4.2 *Refined translation*

Several refinements to the translation in figure 4 are possible. We consider improvements for conditionals, for primitive operations, and for the let expression. In addition, we consider structuring the state and the treatment of large or infinite sets of states.

$$
\begin{aligned}
&|(\texttt{if}\ \ e_1\ \ e_2\ \ e_3)|'_o\overline{@}\langle\overline{S},d,\sigma\rangle\\
&\quad=\ \ \overline{\texttt{let}}\ \langle\overline{S}_1,d_1,\sigma_1,y_1\rangle=|e_1|'_o\overline{@}\langle\overline{S},d,\sigma\rangle\ \overline{\texttt{in}}\\
&\qquad\ \ \underline{\texttt{let}}\ (\overline{S}',\sigma',y')=\\
&\qquad\quad\ \underline{\texttt{if}}\ {}^{(\cup,\phi,\phi)}y_1\\
&\qquad\quad\ \underline{\texttt{then}}\ \overline{\texttt{let}}\ \langle\overline{S}_2,d_2,\sigma_2,y_2\rangle=|e_2|'_o\overline{@}\langle\overline{S}_1,d_1,\sigma_1\rangle\ \overline{\texttt{in}}\ \langle\overline{S}_2,d_2\overline{@}(\sigma_2),y_2\rangle\\
&\qquad\quad\ \underline{\texttt{else}}\ \overline{\texttt{let}}\ \langle\overline{S}_3,d_3,\sigma_3,y_3\rangle=|e_3|'_o\overline{@}\langle\overline{S}_1,d_1,\sigma_1\rangle\ \overline{\texttt{in}}\ \langle\overline{S}_3,d_3\overline{@}(\sigma_3),y_3\rangle\\
&\qquad\ \ \underline{\texttt{in}}\ \langle\overline{S}',\overline{\lambda}\sigma.\sigma,\sigma',y'\rangle
\end{aligned}
$$

Fig. 5. Refined translation: conditional.

$$
\begin{aligned}
&|\texttt{O}(e)|'_o\overline{@}\langle\overline{S},d,\sigma\rangle\\
&\quad=\ \ \overline{\texttt{let}}\ \langle\overline{S}',d',\sigma',y\rangle=|e|'_o\overline{@}\langle\overline{S},d,\sigma\rangle\ \overline{\texttt{in}}\\
&\qquad\ \ \overline{\texttt{let}}\ \overline{S}''=\overline{\delta}(\texttt{O})(\overline{S}')\ \overline{\texttt{in}}\\
&\qquad\ \ \overline{\texttt{let}}\ d''=(\lambda(\sigma)\delta(\texttt{O},y,\sigma))\,\overline{\circ}\,d'\ \overline{\texttt{in}}\\
&\qquad\ \ \overline{\texttt{if}}\ \mathit{bad}\notin\overline{S}''\ \overline{\texttt{then}}\ \underline{\texttt{let}}\ y'=\underline{O}(y)\ \underline{\texttt{in}}\ \langle\overline{S}'',d'',\sigma',y'\rangle\ \overline{\texttt{else}}\\
&\qquad\ \ \underline{\texttt{let}}\ \sigma''=d''\overline{@}(\sigma')\ \underline{\texttt{in}}\\
&\qquad\ \ \underline{\texttt{if}}\ \sigma''\underline{=}\texttt{lift}\ \mathit{bad}\ \underline{\texttt{then}}\ \underline{\texttt{halt}}\ \underline{\texttt{else}}\\
&\qquad\ \ \underline{\texttt{let}}\ y'=\underline{O}(y)\ \underline{\texttt{in}}\\
&\qquad\ \ \langle\overline{S}''\setminus\{\mathit{bad}\},\overline{\lambda}\sigma.\sigma,\sigma'',y'\rangle
\end{aligned}
$$

Fig. 6. Refined translation: primitive operations.

4.2.1 Conditionals

The translation of conditionals can be improved by using techniques from parameterized partial evaluation (Consel & Khoo, 1993). If there is a lattice of compile-time values which only supply approximate information (as is the case here with $\overline{S}\in\mathscr{P}(\overline{State})$) then the dynamic conditional can propagate the compile-time information by taking the join of the values from the branches as it is shown in Fig. 5. The annotation $(\cup,\phi,\phi)$ on the $\underline{\texttt{if}}$ declares this behavior. Here, $\cup$ is the join operation on the lattice $\mathscr{P}(\overline{State})$ and ϕ is the trivial join operator on dynamic values. The annotation follows the type (a triple of values) and specifies one join operator for each component. Otherwise, the translation is analogous to the previous one. In particular, the run-time state σ must also be updated because the transition functions d_i cannot be returned through the dynamic conditional. This behavior amounts roughly to renumbering states as suggested by Colcombet & Fradet (2000).

It is possible to improve further on this behavior by starting from a translation to continuation-passing style, but this improvement can lead to code duplication.

4.2.2 Primitive operations

Another possible improvement lies in the definition of $\overline{\delta}$ in the previous section 4.1. It leads to very conservative results if the state resulting from the application of a primitive operator depends on the actual arguments. For example, consider a refined `read` primitive, `rread`, in the running example. Suppose that sending after reading

$$
\begin{array}{lcl}
|\texttt{let}\ x = e_1\ \texttt{in}\ e_2|'_o \overline{@} \langle \overline{S}, d, \sigma\rangle & = & \overline{\texttt{let}}\ \langle \overline{S}_1, d_1, \sigma_1, x\rangle = |e_1|'_o \overline{@} \langle \overline{S}, d, \sigma\rangle\ \overline{\texttt{in}} \\
& & |e_2|'_o \overline{@} \langle \overline{S}_1, d_1, \sigma_1\rangle
\end{array}
$$

Fig. 7. Refind translation: let expressions

is disallowed only if sensitive files have been read, that is,

$$\delta(\texttt{rread}, \mathit{file}, \mathit{before\text{-}read}) := \begin{cases} \mathit{after\text{-}read} & \text{if } \mathit{sensitive}\ (\mathit{file}) \\ \mathit{before\text{-}read} & \text{otherwise.} \end{cases}$$

In this case, $\overline{\delta}(\texttt{rread})(\mathit{before\text{-}read}) = \mathit{bad}$, so that every `rread` operation in the translated term comes with a run-time check. However, these run-time checks are superfluous because $\delta(\texttt{rread}, \mathit{file}, \sigma)$ is never equal to *bad*.

To address this problem, we redefine $\overline{\delta}$ so that it is no longer a function but rather a binary relation on *State*:

$$(\sigma, \sigma') \in \overline{\delta}(\texttt{O}) \quad \Leftrightarrow \quad \exists y_1 \ldots y_n.\delta(\texttt{O}, y_1, \ldots, y_n, \sigma) = \sigma' \tag{2}$$

Using the relation $\overline{\delta}(\texttt{O})$ as a function $\mathscr{P}(\mathit{State}) \to \mathscr{P}(\mathit{State})$, we now compute with

$$\overline{\delta}(\texttt{O})(\overline{S}) := \{\overline{\sigma}' \mid \overline{\sigma} \in \overline{S}, (\sigma, \sigma') \in \overline{\delta}(\texttt{O})\}$$

the set of all possible next states after being in a state $\sigma \in \overline{S}$. As before, this is a static computation and if the resulting set does not contain *bad*, then it is safe to assume that the state after the operation is not *bad*. The composition of the transition function d is a little bit more complicated. It cannot be completely static anymore because $\overline{\delta}$ is now a relation, that is, it only yields approximate information. Hence, the translation composes d with the dynamic transition function δ for the operation O *and the current arguments y*.

In consequence, the representation of d changes slightly. The implementation encodes it as a list of pairs of primitive operators and their run-time arguments in reverse order of execution. The meaning of the list $(\texttt{O}_n, \vec{y}_n) \ldots (\texttt{O}_1, \vec{y}_1)$ is the composition of the associated state transition functions $\overline{\lambda}\sigma.\delta(\texttt{O}_n, \vec{y}_n, \sigma)\overline{\circ} \ldots \overline{\circ}\overline{\lambda}\sigma.\delta(\texttt{O}_1, \vec{y}_1, \sigma)$ and the empty list stands for $\overline{\lambda}\sigma.\sigma$. As before, this representation enables simple computation of $d(\sigma)$ regardless of σ's binding time and is amenable to memoization.

4.2.3 Handling let expressions.

In the definition of the source language (section 2.1), we have regarded let expressions as a derived notation for an application of a lambda abstraction. However, our translation assumes that each function application involves a non-trivial change in the flow of control. Hence, it obtains unnecessarily bad results for let expressions because it flushes the compile-time information at each let expression.

The remedy is to extend the translation by handling let expressions directly in the obvious way as indicated in figure 7.

4.2.4 Structuring the security state

Many works advise to structure the security state in several independent components (Erlingsson & Schneider, 1999, 2000). This kind of structuring is also possible in our approach. A standard feature of partial evaluators, partially static structures, enables us to model the security state as a compile-time tuple of components that may have independent binding times. For example, there might be static components for small finite sets and dynamic components for elements of large or infinite sets. This structure is also imposed on the static approximation $\overline{S}$.

The introduction of structured states does not require any conceptual change in our translations. Hence, we are not giving new code fragments.

4.2.5 Large and infinite sets of states

For effective use of $\overline{S}$ in the translation, it is vital that it has a finite representation. This is guaranteed if the set *State* is finite. However, it is not essential that $\overline{S}$ contains exact information. Any conservative approximation, that is, any set, which contains the set of possible current states, will do. This is also the key to handle large or even infinite sets of states: we need to design approximations that have finite representations. Technically, what we have specified in section 4.1 is the collecting interpretation of the run-time state $d\overline{@}\sigma$ and the relation $\overline{\delta}$ (from section 4.2.2) is the most accurate approximation of δ. Constructing further approximations is a standard task in abstract interpretation, with many examples in the literature (Cousot & Cousot, 1977; Jones & Nielson, 1995), so we do not go into this here.

4.3 Properties of the staged translation

This section states the properties of the translation in figure 4. The same properties also hold for the refined translation (figures 5, 6 and 7).

The translation is type preserving and fulfils the well-formedness restriction of partial evaluation. In other words, the translation is binding-time analyzed. The judgement $\Gamma \vdash_{bta} e : \tau$ states the latter fact in the form of a two-level type system (Jones *et al.*, 1993).

Proposition 4.1

1. If $\Gamma \vdash_{bta} e : \tau$ then $||\Gamma||_o \vdash |e|_o : |\tau|_o$.
2. If $\Gamma \vdash_{bta} v : \tau$ then $||\Gamma||_o \vdash ||v||_o : ||\tau||_o$.

Next, we prove a sequence of results analogous to the ones in section 3. The only difference is that we first have to erase the newly introduced two-level annotations. The proof techniques are again analogous to those in section 3.2.

A translated expression is safe with respect to $\mathscr{S}'$ (see section 3) and σ.

Proposition 4.2

Let $S \subseteq State \setminus \{bad\}$ with $S \neq \emptyset$ and let $\sigma \in S$.

If $erase(|e|_o)@(S, \lambda\sigma.\sigma, \sigma) \downarrow^{t'} (S', d', \sigma', v')$ then $\delta'^*(t', \sigma) = d'(\sigma')$ where $d'(\sigma') \neq bad$.

If the original term delivers a result without entering a bad state then so does the translated term. In the case of a first-order result, the values agree. Otherwise, they are related by $||\cdot||_o$.

Proposition 4.3
Let $S \subseteq State \setminus \{bad\}$ with $S \neq \emptyset$, $\sigma \in S$ and $\sigma' \neq bad$. Suppose $e \downarrow^t v$ and $\sigma' = \delta^*(t, \sigma)$.

Then $erase(|e|_o)@(S, \lambda\sigma.\sigma, \sigma) \downarrow^{t'} (S', d', \sigma'', erase(||v||_o))$, so that $d'(\sigma'') = \sigma'$ and $t' = |t|\sigma$.

If evaluation of the translated term leads to non-termination or to an undefined primitive operation then so does evaluation of the source term.

Proposition 4.4
If there exists no σ', v', and t' such that $erase(|e|_o)@(S, \lambda\sigma.\sigma, \sigma) \downarrow^{t'} (S', d', \sigma', v')$ then there exists no v and t such that $e \downarrow^t v$.

The safety of the translated program specialized with respect to an initial state σ_0 follows directly from the MIX-equation (1). In our application, there are two compile-time inputs, the program, e, and the initial value for $\overline{\delta}$, and one run-time input, the initial value for σ.

Proposition 4.5
Let `trans` be the two-level term so that $[\![\texttt{trans}]\!]e = |e|_o$, e a closed expression in the source language, $S \subseteq State$ with $bad \notin S$, and $\sigma_0 \in S$.

$$\begin{aligned} &[\![[\![\texttt{spec}]\!]\ (\overline{\texttt{let}}\ \langle S', d', \sigma', v\rangle = \texttt{trans}\overline{@}e\overline{@}\langle S, \lambda(\sigma)\sigma, \underline{\sigma_0}\rangle\ \overline{\texttt{in}}\ v)]\!] \\ = \quad &\texttt{let}\ (S', s', \sigma', v) = erase(|e|_o)@(S, \lambda\sigma.\sigma, \sigma_0)\ \texttt{in}\ v \end{aligned}$$

Here, $\underline{\sigma_0}$ is the dynamic "lifted" value corresponding to σ_0.

The safety of the thus compiled program follows from the MIX-equation (1) and from Proposition 4.5.

Corollary 4.6
The compiled program

$$[\![\texttt{spec}]\!]\ (\overline{\texttt{let}}\ \langle S', d', \sigma', v\rangle = \texttt{trans}\overline{@}e\overline{@}\langle S, \lambda(\sigma)\sigma, \underline{\sigma_0}\rangle\ \overline{\texttt{in}}\ v)$$

is safe with respect to $\mathscr{S}'$ (as defined in section 3).

4.4 Non-functional properties

We have claimed that a partial evaluator running the staged translations in figures 4–7 will translate a program in linear time and that the size of the translated program will be linear, both in the size of the original program. Technically, we prove the following claims, where n is the size of the translated expression and $s = |State|$:

- The translation runs in time $O(n \cdot s^2)$.
- The size of the translated code is in $O(n)$.
- As a corollary of the latter, we obtain that the translation does not duplicate code.

The following assumptions are used to justify this claim:

- The partial evaluator employs context propagation, that is, it floats dynamic let expressions out of the way of compile-time computations. This is a standard feature of partial evaluators for functional programming languages. The consequence of context propagation is that the body of a dynamic let expression can return compile-time values (Bondorf, 1992).
- Contexts are *not* propagated to the branches of dynamic conditionals. This feature was suggested by Lawall & Danvy (1994), but dismissed because it may lead to code duplication.
- Dynamic conditionals may return approximate compile-time values (only used for parameterized partial evaluation in section 4.2).
- An application of the transition function δ takes constant time.

We have used the author's PGG system for the Scheme language in our experiments (Thiemann, 2000, 1999a). The released version of the system contains the features listed.

To support the claim that the translation runs in time $O(n \cdot s^2)$, we examine each case of the translation in figure 4.

- x: constant time.
- a: constant time.
- `fix`: all operations in the translated term take constant time except the computation of $\overline{\delta}(\sigma')$. Since $\overline{\delta}$ is represented by a list of operations, this last step may take time proportional to the number of operations in the body of the function. At this point, we require a simple amortization argument: Since each list of operations is flushed in this way at most once, we distribute the time taken for applying $\overline{\delta}$ by adding a constant-time step to each primitive operation.
- For an n-tuple, only the tuple must be created in constant time (assuming that there is an upper bound on the size of tuples created in a program).
- The embedding of a value takes constant time.
- `if`: all operations take constant time except the computations of $\overline{\delta}_2(\sigma_2)$ and $\overline{\delta}_3(\sigma_3)$, each of which is again dealt with by the amortization argument.
- Tuple computation: all operations take constant time.
- Primitive operation:
  - — The computation of $\overline{\delta}(\mathtt{O})(\overline{S}')$ can take time quadratic in the size of *State* (assuming that the set is implemented as a boolean vector and $\overline{\delta}$ is represented by a boolean matrix the translation must multiply the matrix with the vector).
  - — Composing the transition functions as in $\overline{\delta}(\mathtt{O}) \mathbin{\overline{\circ}} d'$ takes constant time.
  - — The static text $bad \notin \overline{S}''$ takes time linear in $|State|$, which is again dominated by the quadratic time bound above.
  - — The remaining expressions are run-time computations, so their construction takes constant time.
  - — For technical reasons, we need to spend one unit of constant time, which we lend to at most one use of flushing in the transformation.

- Function application: The only non-trivial computation is $d_1\overline{@}(\sigma_1)$, which is dealt with using our amortization argument.

In summary, the time spent at each of the n subexpressions of the source program is bounded by $O(s^2)$, so that the time spent for translating the source program is $O(n \cdot s^2)$.

To conclude that the translation produces code whose size is linear in the size of the input code, we recall the following facts:

- The translation takes linear time to run.
- The output code is a tree, i.e. there is no sharing of output program fragments.
- The operations working in time quadratic in $|State|$ are purely compile-time, they do not generate any code.

The claim is immediate from these facts.

As a corollary, we find that no code duplication can occur. In the presence of code duplication, it would not be possible to prove a linear size bound.

Both claims still hold for our refined translations of conditionals and primitive operations. In the case of the conditional, only the $\cup$ operation annotating the conditional in figure 5 can make a difference with respect to our analysis. It can take quadratic time in $|State|$. Hence, it is also dominated by the assumed quadratic time bound.

The analysis for primitive operations (figure 6) is identical to the previous one.

5 Conclusions

Offline partial evaluation is well-suited to translate programs into secured programs that comply with security policies specified by security automata. We have demonstrated this using a simple homogeneous translation to have a clear presentation of the fundamental concepts. More run-time checks could be eliminated by considering a number of avenues, for example, by using stronger specialization techniques (Thiemann, 2001), by allowing for code duplication, or by flow analysis (Jagannathan & Wright, 1998).

The techniques shown here are designed to

- never duplicate code,
- run efficiently and predictably in time linear in the size n of the input ($O(n \cdot s^2)$ where s is the number of states of the security automaton), and
- be applicable with all specialization techniques, e.g. online partial evaluation (Weise *et al.*, 1991), offline partial evaluation (Jones *et al.*, 1993), type specialization (Hughes, 1996), type-directed partial evaluation (Danvy, 1996).

The translations presented in this work have been designed and tested using the PGG system (Thiemann, 2000), an offline partial evaluator for Scheme. Since this system relies on combinators to generate specialized programs (Thiemann, 1999b), only these combinators need to be part of the TCB.

The factor $O(s^2)$ above looks daunting. We rely on suitable abstractions (see section 4.2.5) of the set of run-time states to a tractable number of compile-time states, so that s is never large and we can regard $O(s^2)$ as a constant.

On straight-line code, the results of the refined translation are comparable to those of Colcombet & Fradet (2000). For programs that involve function calls, their technique yields better results because our translation does not transfer information across function calls. Basing the translation on the results of a flow analysis will improve on that. However, this option means to increase the size of the trusted computing base and it is rejected by Erlingsson & Schneider (2000) for that reason.

Only extensive practical experience can show whether the number of run-time tests remaining in transformed programs is acceptable. For that reason, we plan to integrate the translation with run-time code generation and conduct such tests. The resulting framework will provide just-in-time securing compilation and it will serve as a vehicle for experiments with mobile code.

Acknowledgements

Many thanks to the reviewers for their extensive and thorough comments. Their remarks have lead to considerable improvements in the presentation of the work.

References

Adl-Tabatabai, A.-R., Langdale, G., Lucco, S. and Wahbe, R. (1996) Efficient and language-independent mobile programs. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. Philadelphia, PA. ACM Press.

Alpern, B. and Schneider, F. B. (1985) Defining liveness. *Infor. Process. Lett.* **21**(4), 181–185.

Alpern, B. and Schneider, F. B. (1987) Recognizing safety and liveness. *Distributed Comput.* **2**, 117–126.

Appel, A. W. and Felty, A. P. (2000) A semantics model of types and machine instructions for proof-carrying code. In: Reps, T., editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*. Boston, MA. ACM Press.

Bondorf, A. (1992) Improving binding times without explicit CPS-conversion. *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pp. 1–10.

Colcombet, T. and Fradet, P. (2000) Enforcing trace properties by program transformation. In: Reps, T., editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*. Boston, MA. ACM Press.

Consel, C. and Danvy, O. (1993) Tutorial notes on partial evaluation. *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pp. 493–501. Charleston, SC. ACM Press.

Consel, C. and Khoo, S. C. (1991) Parameterized partial evaluation. *Proc. Conference on Programming Language Design and Implementation*. Toronto, Canada. ACP Press.

Consel, C. and Khoo, S. C. (1993) Parameterized partial evaluation. *ACM Trans. Program. Lang. Syst.* **15**(3), 463–493.

Consel, C. and Khoo, S. C. (1995) On-line and off-line partial evaluation: semantic specifications and correctness proofs. *J. Functional Program.* **5**(4), 461–500.

Cousot, P. and Cousot, R. (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proc. 4th Annual ACM Symposium on Principles of Programming Languages*, pp. 238–252. ACM.

Crary, K., Walker, D. and Morrisett, G. (1999) Typed memory management in a calculus of capabilities. In: Aiken, A., editor, *Proc. 26th Annual ACM Symposium on Principles of Programming Languages*, pp. 262–275. San Antonio, TX. ACM Press.

Danvy, O. (1996) Type-directed partial evaluation. *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 242–257. St. Petersburg, FL. ACM Press.

Erlingsson, Ú. and Schneider, F. B. (1999) SASI enforcement of security policies: A retrospective. *Proceedings New Security Paradigms Workshop.*

Erlingsson, Ú. and Schneider, F. B. (2000) IRM enforcement of Java stack inspection. *IEEE Symposium on Security and Privacy*. Oakland, CA. IEEE Computer Society, California.

Evans, D. and Twyman, A. (1999) Flexible policy-directed code safety. *IEEE Symposium on Security and Privacy.*

Flanagan, C., Sabry, A., Duba, Bruce F. and Felleisen, M. (1993) The essence of compiling with continuations. *Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 237–247.

Glück, R. (1994) On the generation of specializers. *J. Functional Program.* **4**(4), 499–514.

Glück, R. and Jørgensen, J. (1994a) Generating optimizing specializers. *IEEE International Conference on Computer Languages 1994*, pp. 183–194. Toulouse, France. IEEE Press.

Glück, R. and Jørgensen, J. (1994b) Generating transformers for deforestation and supercompilation. In: Le Charlier, B., editor, *Static Analysis: Lecture Notes in Computer Science 864*, pp. 432–448. Springer-Verlag.

Hatcliff, J. (1995) Mechanically verifying the correctness of an offline partial evaluator. In: Swierstra, D. and Hermenegildo, M., editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95): Lecture Notes in Computer Science 982*, pp. 279–298. Springer-Verlag.

Hatcliff, J. and Danvy, O. (1994) A generic account of continuation-passing styles. *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pp. 458–471. Portland, OR. ACM Press.

Hatcliff, J. and Danvy, O. (1997) A computational formalization for partial evaluation. *Math. Struct. Comput. Sci.* **7**(5), 507–542.

Hughes, J. (1996) Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In: Danvy, O., Glück, R. and Thiemann, P., editors, *Partial Evaluation: Lecture Notes in Computer Science 1110*, pp. 183–215. Schloß Dagstuhl, Germany. Springer-Verlag.

J2SE (2000) *Java2 platform.* http://www.javasoft.com/products/.

Jagannathan, S. and Wright, A. (1998) Polymorphic splitting: An effective polyvariant flow analysis. *ACM Trans. Program. Lang. Syst.*, **20**(1), 166–207.

Jones, N. D. (1997) Combining abstract interpretation and partial evaluation (brief overview). In: Van Hentenryck, P., editor, *Proc. International Static Analysis Symposium, SAS'97: Lecture Notes in Computer Science 1302*, pp. 396–405. Berkeley, CA. Springer-Verlag.

Jones, N. D. and Nielson, F. (1995) Abstract interpretation. In: Abramsky, S., Gabbay, D. and Maibaum, T. S. E., editors, *Handbook of Logic in Computer Science*, vol. 4. Oxford University Press.

Jones, N. D., Gomard, C. K. and Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generation.* Prentice-Hall.

Kishon, A., Hudak, P. and Consel, C. (1991) Monitoring semantics: A formal framework for specifying, implementing and reasoning about execution monitors. *Proc. Conference on Programming Language Design and Implementation.* Toronto, Canada. ACP Press.

Kozen, D. (1999) *Language-based security*. Technical Report TR99-1751, Cornell University, Computer Science.

Launchbury, J. and Holst, C. K. (1991) Handwriting cogen to avoid problems with static typing. *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pp. 210–218.

Lawall, J. L. and Danvy, O. (1994) Continuation-based partial evaluation. *Proc. 1994 ACM Conference on Lisp and Functional Programming*, pp. 227–238. Orlando, FL. ACM Press.

Lawall, J. L. and Thiemann, P. (1997) Sound specialization in the presence of computational effects. *Proc. Theoretical Aspects of Computer Software: Lecture Notes in Computer Science 1281*, pp. 165–190. Sendai, Japan. Springer-Verlag.

Lee, I., Kannan, S., Kim, M., Sokolsky, O. and Viswanathan, M. (1999) Runtime assurance based on formal specifications. *1999 International Conference on Parallel and Distributed Processing Techniques and Applications.*

Lee, P. and Leone, M. (1996) Optimizing ML with run-time code generation. *Proc. Conference on Programming Language Design and Implementation.* Toronto, Canada. ACP Press.

Lindholm, T. and Yellin, F. (1996) *The Java Virtual Machine Specification.* Addison-Wesley.

Lucco, S., Sharp, O. and Wahbe, R. (1995) Omniware: A universal substrate for web programming. *Worldwideweb JL.* **1**(1).

Michael, N. G. and Appel, A. W. (2000) Machine instruction syntax and semantics in higher order logic. *17th International Conference on Automated Deduction* (*CADE-17*).

Morrisett, G., Walker, D., Crary, K. and Glew, N. (1998) From system F to typed assembly language. In: Cardelli, L., editor, *Proc. 25th Annual ACM Symposium on Principles of Programming Languages.* San Diego, CA. ACM Press.

Necula, G. C. (1997) Proof-carrying code. In: Jones, N. D., editor, *Proc. 24th Annual ACM Symposium on Principles of Programming Languages.* Paris, France. ACM Press.

Necula, G. C. and Lee, P. (1998) Safe, untrusted agents using proof-carrying code. In: Vigna, G., editor, *Mobile Agent Security: Lecture Notes in Computer Science 1419*, pp. 61–91. Springer-Verlag.

Plattner, B. and Nievergelt, J. (1981) Monitoring program execution: A survey. *Computer*, **16**(11), 76–93.

PLDI '96 (1996) *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation.* Philadelphia, PA. ACM Press.

PLDI '91 (1991) *Proc. Conference on Programming Language Design and Implementation.* Toronto, Canada. ACM Press.

Reps, T. (ed) (2000) *Proc. 27th Annual ACM Symposium on Principles of Programming Languages.* Boston, MA. ACM Press.

Schneider, F. B. (2000) Enforceable security policies. *ACM Trans. Infor. Syst. Security*, **3**(1), 30–50.

Thiemann, P. (1999a) Aspects of the PGG system: Specialization for Standard Scheme. In: Hatcliff, J., Mogensen, T. Æ. and Thiemann, P., editors, *Partial Evaluation – Practice and Theory. Proceedings DIKU International Summerschool: Lecture Notes in Computer Science 1706*, pp. 412–432. Copenhagen, Denmark. Springer-Verlag.

Thiemann, P. (1999b) Combinators for program generation. *J. Functional Program* **9**(5), 483–525.

Thiemann, P. (2000) *The pgg system—user manual.* Universität Freiburg, Freiburg, Germany. (Available from `http://www.informatik.uni-freiburg.de/proglang/software/pgg/`.)

Thiemann, P. (2001) Enforcing safety properties using type specialization. In: Sands, D., editor, *Proc. 10th European Symposium on Programming: Lecture Notes in Computer Science.* Genova, Italy. Springer-Verlag.

Turchin, V. F. (1993) Program tranformation with metasystem transitions. *J. Functional Program* **3**(3), 283–313.

Wahbe, R., Lucco, S., Anderson, T. E. and Graham, S. L. (1993) Efficient software-based fault isolation. *Proceedings 14th ACM Symposium on Operating Systems Principles*, pp. 203–216.

Walker, D. (2000) A type system for expressive security policies. *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*. Boston, MA. ACM Press.

Weise, D., Conybeare, R., Ruf, E. and Seligman, S. (1991) Automatic online partial evaluation. In: Hughes, J., editor, *Proc. Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 523*, pp. 165–191. Cambridge, MA. Springer-Verlag.