Enforcing Resource Bounds via Static Verification of Dynamic Checks

AJAY CHANDER, DAVID ESPINOSA, and NAYEEM ISLAM
DoCoMo Labs USA
PETER LEE
Carnegie Mellon University
and
GEORGE C. NECULA
University of California, Berkeley

We show how to limit a program's resource usage in an efficient way, using a novel combination of dynamic checks and static analysis. Usually, dynamic checking is inefficient, due to the overhead of checks, while static analysis is difficult and rejects many safe programs. We propose a hybrid approach that solves these problems. We split each resource-consuming operation into two parts. The first part is a dynamic check, called `reserve`. The second part is the actual operation, called `consume`, which does not perform any dynamic checks. The programmer is then free to hoist and combine `reserve` operations. Combining `reserve` operations reduces their overhead, while hoisting `reserve` operations ensures that the program does not run out of resources at an inconvenient time. A static verifier ensures that the program reserves resources before it consumes them. This verification is both easier and more flexible than a priori static verification of resource usage. We present a sound and efficient static verifier based on Hoare logic and linear inequalities. As an example, we present a version of `tar` written in Java.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*Assertion Checkers*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic Storage Management*; D.4.6 [**Operating Systems**]: Security and Protection—*Verification*

General Terms: Languages, Security, Verification

Additional Key Words and Phrases: Resource bounds, Static, Dynamic

1. INTRODUCTION

Users are downloading code to run on their devices — computers, PDAs, cell phones, etc — with increasing frequency. Examples of downloaded code include software updates, applications, games, active web pages, proxies for new protocols, codecs for new formats, and front-ends for distributed applications. At the same time, viruses, worms, and other malicious agents have also become common, resulting in attacks that include data corruption, privacy violation, and denial of service based on overuse of system resources. The latter problem is particularly relevant

Author's address: Ajay Chander, DoCoMo Labs USA, 3240 Hillview Ave, Palo Alto, CA, 94304.

for small devices such as PDAs and cell phones. The state of the practice in mobile code execution includes powerful techniques that prevent data corruption (e.g., bytecode verification), but the enforcement of resource quotas is comparatively less developed. In this paper, we provide an efficient and flexible approach to limiting the resource usage of untrusted code. By *flexible*, we mean that our method applies to all sequential computer programs, including those where resource usage is not known until runtime. By *efficient*, we mean that it enforces resource quotas with significantly fewer runtime checks than previous methods based exclusively on dynamic checking.

In our scenario, the *code consumer* runs an *untrusted* program created by a *code producer*. This program interfaces to the code consumer's computer via a *trusted* runtime library that provides functions to access resources. We consider both physical resources such as CPU, memory, disk, and network, as well as virtual resources such as files, database connections, and processes. Our goal is to limit resource use according to the code consumer's *security policy*. This policy specifies a *quota* for each resource that the program can use.

Our technique enforces resource quotas with a combination of static and dynamic checks. More precisely, we verify statically that a program's dynamic checks are sufficient to ensure that the quota is not exceeded. To support such hybrid checking, we split each resource-using operation into two separate operations, `reserve` and `consume`. `Reserve` performs a dynamic check against the quota, and `consume` actually uses the resource. If `reserve` succeeds, it guarantees that the resource is available, so that `consume` does not need to perform any dynamic checks.

For example, suppose that a runtime library has a function `readNetworkBytes` that reads bytes from the network. We replace it with `reserveNetworkBytes` and `readNetworkBytes`. The former reserves the right to read bytes from the network, while the latter actually performs the I/O.

In this paper, without loss of generality, we consider a single resource, whose reservation operation is called `reserve` and whose consumption operation is called `consume`. In general, the runtime library has several types of resources, and each resource type has several operations that consume it. We assume that the runtime library is part of the trusted computing base (TCB), including the implementations of `reserve` and `consume`.

Current libraries perform `reserve` and `consume` together when a resource is used. It is easy to replace these calls automatically by pairs of separate calls to `reserve` and `consume`. It is also easy to verify statically that the result of this transformation never uses more resources than have been reserved.

The advantage of this separation is that the programmer, or appropriate optimization tools, can combine multiple `reserve`s into one and can hoist `reserve` out of a loop whose body consumes resources. In this paper, we describe a static analysis that verifies that a program's placement of `reserve`s is sufficient. If the resource expressions that occur in the `reserve` and `consume` expressions are linear, then the analysis is decidable and efficient, and our experiments show that it succeeds even in the presence of aggressive optimizations.

Moving `reserve` out of a loop can yield an arbitrary improvement in the number of dynamic checks. This improvement results in significant performance gains if the `reserve` consults a complex or remote resource manager.

```
Program Dynamic
  while read() ≠ 0
    consume 1

Program Static
  i := 0
  while i < 10000
    consume 1
    i := i + 1

Program Mixed1
  N := read()
  i := 0
  while i < N
    consume 1
    i := i + 1

Program Mixed2
  while read() ≠ 0
    i := 0
    while i < 100
      consume 1
      i := i + 1
```

Fig. 1. Example programs

Also, moving checks earlier can guarantee that no resource errors occur in critical code fragments such as atomic transactions. A standard compiler cannot perform this optimization, because it changes the semantics of the program.

Section 2 introduces an imperative language with resource-aware constructs and illustrates the benefits of our method over purely static or dynamic approaches. Section 3 presents an operational semantics for our language and precisely characterizes resource-use safety. Section 5 describes the safety condition generator (SCG) (Section 5) and proves that it is sound. Section 6 shows how to construct a prover that efficiently discharges the safety conditions. Section 7 presents our experience using ESC/Java to check a Java implementation of `tar`. Section 8 positions this paper with respect to relevant work in a few areas, Section 9 mentions ongoing efforts and future work, and Section 10 concludes. The appendices present additional technical information.

2. CONCEPT

Figure 1 shows four programs that use resources. Program *Dynamic* reads a sequence of numbers until it sees a zero. It consumes one resource unit for each number that it reads. Note that `read` does not consume resources; only `consume` does. Program *Static* takes no input and consumes exactly 10000 resource units. Program *Mixed1* reads a number n, then consumes n resource units. Program *Mixed2* reads a sequence of numbers until it sees a zero. It consumes 100 resource units for each number that it reads.

A standard dynamic checking approach performs one check for each `consume`. It executes all four programs safely but adds unnecessary overhead to the *Static* and *Mixed* programs. A typical static analyzer adds no overhead to the *Static* program but cannot execute the other three safely.

We present a method that has the advantages of both static and dynamic checkers. Like the dynamic checker, it safely executes all four programs. Like the static checker, it uses the static information available in each program to run more efficiently.

Figure 2 shows the same four programs with `reserve` operations added. Each `reserve` reserves resources from the runtime system, for later use by `consume`. The `reserve` performs a dynamic check and debits the program's resource quota,

```
Program Dynamic
  while read() ≠ 0
    reserve 1
    consume 1
  inv (true, 0)
```

```
Program Static
  reserve 10000
  i := 0
  while i < 10000
    consume 1
    i := i + 1
  inv (i ≤ 10000, 10000 − i)
```

```
Program Mixed1
  N := read()
  reserve N
  i := 0
  while i < N
    consume 1
    i := i + 1
  inv (i ≤ N, N − i)
```

```
Program Mixed2
  while read() ≠ 0
    reserve 100
    i := 0
    while i < 100
      consume 1
      i := i + 1
    inv (i ≤ 100, 100 − i)
```

Fig. 2. Example programs with reservations.

while `consume` simply performs a resource-using operation; it does *not* perform any dynamic checks. For example, if the resource is file I/O, then `reserve` n checks that the program can write n bytes to the file system, and `consume` actually writes the bytes.

Note that we annotate each `while` loop with an invariant (A, e), which indicates that the predicate A holds and at least e resources have been reserved but not yet consumed. The invariant must hold before the loop test on every iteration. The invariants are necessary to support our static checker. As the *Dynamic* program suggests, non-trivial invariants are only necessary when hoisting reservations out of loops. When we statically check the program's resource use, we also check that its invariants hold.

Note that all four programs reserve enough resources. *Dynamic* performs exactly the same checks that it would in a dynamic system by acquiring each resource just before using it. *Static* performs exactly one check at the very beginning of execution. *Mixed1* and *Mixed2* perform far fewer checks than they would in a dynamic system by reserving resources as early as possible; for example, *Mixed1* performs N fewer checks and *Mixed2* performs 100 times fewer checks.

Note also that program *Dynamic* can run out of resources at any point during its execution. Thus, it must be prepared to handle errors at all intermediate states. On the other hand, program *Mixed1* can only run out of resources at the very beginning. Once it enters the main loop, it is guaranteed to succeeed. Thus, it only needs to handle errors at the beginning, where error recovery is simpler.

To ensure that a program is safe, we need to prove that its resource consumption is less than the quota imposed by the runtime system. Stated informally, we need to prove *consumption* $\leq$ *quota*. Using `reserve`, we factor this condition as *consumption* $\leq$ *reservation*, which we check statically, and *reservation* $\leq$ *quota*, which we check dynamically.

If the resource is reusable, then the arguments to `reserve` and `consume` are allowed to be negative. A negative `consume` frees resources for later use. For example, `malloc` has positive consumption, but `free` has negative consumption. A

negative `reserve` returns resources to the runtime system when the program no longer needs them.

This paper focuses on manual resource management, where the programmer manually obtains resources, and manually returns them, if they are reusable. A garbage collector can perform automated reclamation, but most collectors are dynamic, and thus do not benefit from the static aspects of our technique. However, an automated annotator can statically insert `reserve` where necessary, so that the resulting program successfully passes the verifier.

3. LANGUAGE

To formalize the static checking procedure, we use a simple imperative programming language that computes with integer values. We assume that there is one resource of interest whose amount is measured in some arbitrary unit. The command `consume` e models any runtime operation that consumes e resource units, where e is an expression in the language. The command `reserve` e reserves e resource units from the runtime system. The reservation may fail, but if it succeeds, we know that e resource units have been reserved.

Figure 3 shows the syntax of the full language. We assume that the variables x take only integer values. Expressions are linear in their variables, since we can multiply an expression by a constant but not by another expression.

The expression $\texttt{cond}(b, e_1, e_2)$ has value e_1 if b has value `true` and e_2 if b has value `false`. The predicate $\texttt{cond}(b, P_1, P_2)$ has value P_1 if b has value `true` and P_2 if b has value `false`. The propositional connectives $\wedge$, $\forall$, and $\Rightarrow$ have their usual meanings.

In addition to the structured control provided by `while`, we provide unstructured control via functions. We assume that functions pass arguments and return results in global variables. We annotate each function with a pre condition labelled `req` ("requires") and a post condition labelled `ens` ("ensures"). These conditions provide an interface between the caller and callee. At a function call, we *prove* the pre condition and *assume* the post condition. In the function body, we *assume* the pre condition and *prove* the post condition.

We syntactically distinguish the annotations A, a subset of the predicates P. Annotations are conjunctions of equalities and inequalities between expressions.

p	$::=$	$f^* c$	Programs
f	$::=$	$\texttt{fun}\ f\ \texttt{req}\ (A_0, e_0)\ \texttt{ens}\ (A_1, e_1)\ c\ \texttt{end}$	Functions
c	$::=$	$\texttt{skip} \mid x := e \mid c_1; c_2 \mid$	
		$\texttt{consume}\ e \mid \texttt{reserve}\ e$	
		$\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid$	
		$\texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (A, e) \mid$	
		$\texttt{call}\ f$	Commands
e	$::=$	$x \mid n \mid e_1 + e_2 \mid n * e \mid \texttt{cond}(b, e_1, e_2)$	Integer expressions
b	$::=$	$\texttt{true} \mid e_1 \geq e_2 \mid e_1 = e_2$	Boolean expressions
P	$::=$	$b \mid P_1 \wedge P_2 \mid A \Rightarrow P \mid$	
		$\forall x.P \mid \texttt{cond}(b, P_1, P_2)$	Predicates
A	$::=$	$b \mid A_1 \wedge A_2$	Annotations

Fig. 3. Simple imperative language definition.

```
Program Nonlinear1
  N := read()
  reserve N * N
  i := 0
  while i < N
    j := 0
    while j < N
      consume 1
      j := j + 1
    inv (j ≤ N, N * N − i * N − j)
    i := i + 1
  inv (i ≤ N, N * N − i * N)
```

```
Program Nonlinear2
  N := read()
  i := 0
  while i < N
    reserve N
    j := 0
    while j < N
      consume 1
      j := j + 1
    inv (j ≤ N, N − j)
    i := i + 1
  inv (true, 0)
```

Fig. 4. Example nonlinear programs.

The predicates P also allow universal quantification, conditionals, and implication, but the left side of an implication is an annotation. These restrictions allow us to define a simple prover based on a standard decision procedure for equality and linear arithmetic (see Section 6).

Loop invariants and function pre and post conditions are annotations, not full predicates. Since the `scg` only assumes pre conditions (at the start of a method), post conditions (at function call return), loop invariants (at the start of the loop body), and conditional tests (in each branch of the conditional), it is clear that the left side of each implication is an annotation.

We can generalize the language in several ways. First, we can distinguish program expressions from resource expressions and allow program expressions to be nonlinear. Second, we can strengthen the prover to include quadratic or higher polynomial resource expressions. In Figure 4, we could then write program *Nonlinear1* instead of program *Nonlinear2*. Third, we can add negation and disjunction to the P predicates with only a small loss in efficiency.

Our earlier paper [Chander et al. 2005] describes how to write an SCG for Java bytecode. To handle unstructured code, we require the programmer to supply an invariant at the target of each backward branch. We verify each code path separately, where a code path begins at an invariant (or the start of a method) and ends at an invariant (or the end of a method). Along each code path, we use essentially the same technique described here.

Our language does not include parallelism or multiple threads. Addding these features complicates the verifier considerably, since it has to account for all possible interleaved thread executions. This complexity distracts from our main point, the flexible combination of static and dynamic resource management.

4. SEMANTICS

This section formalizes the meaning of expressions and commands using a standard operational semantics [Mitchell 1996]. The execution state is a pair $\langle \sigma, n \rangle$ of an environment σ that maps variable names to integer values and a natural number n that represents the amount of *available* resources, that is, resources that have been reserved but not yet consumed. When we modify n, we carefully ensure that it is non-negative.

$$\frac{}{\langle \texttt{skip}, \sigma, n\rangle \Downarrow \langle \sigma, n\rangle}\ \text{SKIP} \qquad \frac{\langle c', \sigma, n\rangle \Downarrow \langle \sigma', n'\rangle \quad \langle c'', \sigma', n'\rangle \Downarrow R}{\langle c'; c'', \sigma, n\rangle \Downarrow R}\ \text{SEQ}$$

$$\frac{\langle c', \sigma, n\rangle \Downarrow \mathit{Error}}{\langle c'; c'', \sigma, n\rangle \Downarrow \mathit{Error}}\ \text{SEQE} \qquad \frac{}{\langle x := e, \sigma, n\rangle \Downarrow \langle \sigma[x := [\![e]\!]\sigma], n\rangle}\ \text{ASSIGN}$$

$$\frac{\langle \sigma, n\rangle \models e}{\langle \texttt{consume}\ e, \sigma, n\rangle \Downarrow \langle \sigma, n - [\![e]\!]\sigma\rangle}\ \text{C-OK} \qquad \frac{\langle \sigma, n\rangle \not\models e}{\langle \texttt{consume}\ e, \sigma, n\rangle \Downarrow \texttt{ReservationExceeded}}\ \text{C-FAIL}$$

$$\frac{\langle \sigma, n\rangle \models -e}{\langle \texttt{reserve}\ e, \sigma, n\rangle \Downarrow \langle \sigma, n + [\![e]\!]\sigma\rangle}\ \text{R-OK} \qquad \frac{\langle \sigma, n\rangle \not\models -e}{\langle \texttt{reserve}\ e, \sigma, n\rangle \Downarrow \texttt{ReservationExceeded}}\ \text{R-FAIL1}$$

$$\frac{}{\langle \texttt{reserve}\ e, \sigma, n\rangle \Downarrow \texttt{ReserveFailed}}\ \text{R-FAIL2}$$

$$\frac{\sigma \models b \quad \langle c, \sigma, n\rangle \Downarrow R}{\langle \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c', \sigma, n\rangle \Downarrow R}\ \text{IFT} \qquad \frac{\sigma \not\models b \quad \langle c', \sigma, n\rangle \Downarrow R}{\langle \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c', \sigma, n\rangle \Downarrow R}\ \text{IFF}$$

$$\frac{\sigma \not\models b}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (A, e), \sigma, n\rangle \Downarrow \langle \sigma, n\rangle}\ \text{WHILEF}$$

$$\frac{\sigma \models b \quad \langle c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (A, e), \sigma, n\rangle \Downarrow R}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (A, e), \sigma, n\rangle \Downarrow R}\ \text{WHILET}$$

$$\frac{\texttt{fun}\ f\ \texttt{req}\ (A_0, e_0)\ \texttt{ens}\ (A_1, e_1)\ c\ \texttt{end} \quad \langle c, \sigma, n\rangle \Downarrow R}{\langle \texttt{call}\ f, \sigma, n\rangle \Downarrow R}\ \text{FUN}$$

Fig. 5. Operational semantics

We write $[\![e]\!]\sigma$, $[\![b]\!]\sigma$, $[\![A]\!]\sigma$, and $[\![P]\!]\sigma$ for the values of e, b, A, and P in the environment σ. For example,

$$[\![\texttt{cond}(b, e_1, e2)]\!]\sigma = \begin{cases} [\![e_1]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{true} \\ [\![e_2]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \end{cases}$$

We use the notation $\sigma[x := n]$ to denote the environment that is identical to σ except that x is set to n. We write $\sigma \models P$ if $[\![P]\!]\sigma = \texttt{true}$, and similarly for $\sigma \models A$ and $\sigma \models b$. We write $\langle \sigma, n\rangle \models e$ if $n \geq [\![e]\!]\sigma$. We write $\langle \sigma, n\rangle \models (P, e)$ if $\sigma \models P$ and $\langle \sigma, n\rangle \models e$, and similarly for $\langle \sigma, n\rangle \models (A, e)$. In general, we write $x \not\models y$ if it is not the case that $x \models y$.

We define the operational semantics of our language in terms of the judgment $\langle c, \sigma, n\rangle \Downarrow R$, which means that the evaluation of command c starting in state $\langle \sigma, n\rangle$ terminates with result R. The result R can be one of the following types of values. If the command terminates normally, then R is a new state $\langle \sigma', n'\rangle$. If a reservation fails, then R is the error `ReserveFailed`. If the program uses more resources than it has reserved, then R is the error `ReservationExceeded`. We use the meta-variable *Error* to stand for one of these two errors.

Figure 5 shows the operational semantics for our language. The `consume` and `reserve` operations maintain the invariant that the amount n of available resources

is non-negative. The program can potentially violate this invariant when allocating resources by calling `consume` with a positive argument or when returning resources to the runtime system by calling `reserve` with a negative argument. In both cases, we generate the `ReservationExceeded` error if the available resources would otherwise become negative.

The `reserve` operation can also nondeterministically yield a `ReserveFailed` error. This behavior models the case when the program's runtime quota is exceeded. A deterministic semantics could add an explicit counter for the resources available in the runtime system.

5. SAFETY CONDITION GENERATOR

This section defines the safety condition generator `scg` and proves a soundness theorem, which says that if the safety condition is valid, then the program is actually safe. The `scg` uses a variant of Dijkstra's weakest precondition calculus [Dijkstra 1976]. We adapt the calculus to use "generalized predicates" (P, e), which abbreviate $P \wedge n \geq e$, meaning that P holds and at least e resource units have been reserved but not yet consumed. The generalized predicate notation lets us increment and decrement e explicitly, instead of using substitution on P.

The `scg` takes a command and a post condition and produces a pre condition. The pre and post conditions are both generalized predicates. The soundness theorem shows that if the pre condition produced by `scg` holds before c runs, and the `ReserveFailed` error does not occur, then the post condition holds after. It also shows that the `ReservationExceeded` error cannot occur.

To show that a program $p = f_1 \ldots f_n c$ is safe to execute, we prove its verification condition $\texttt{scg}_{prog}(p)$, which is the conjunction of $\texttt{scg}_{cmd}(c)$ with $\texttt{scg}_{fun}(f_i)$ for each f_i. The function $\texttt{scg}_{fun}(f)$ verifies f with respect to its specified pre and post conditions. The function $\texttt{scg}_{cmd}(c)$ verifies c with respect to the pre condition $(\texttt{true}, 0)$ and the post condition $(\texttt{true}, 0)$. Since all states satisfy the generalized predicate $(\texttt{true}, 0)$, the actual point is to ensure that the error `ReservationExceeded` does not occur. Thus, the runtime system does not check for it, nor does it maintain the counter n. Note that the runtime system *does* check for the `ReserveFailed` error, which indicates that the program has dynamically exceeded its resource quota.

As Figure 6 shows, we define `scg` by induction on the syntax of commands. If we interpret the generalized predicate (P, e) as the standard predicate $P \wedge n \geq e$, where n is the amount of resources reserved but not yet consumed, then our definition matches the standard definition of weakest precondition for assignment, sequencing, conditionals, loops, and functions.

We computed the `scg` for `reserve` and `consume` from the proof of the soundness theorem (see the Appendix). However, we can think of `reserve` and `consume` as the following commands:

$$\texttt{reserve}\ e' \ = \ \texttt{if}\ n + e' \geq 0\ \texttt{then}\ n := n + e'\ \texttt{else}\ \texttt{ReservationExceeded}$$
$$\texttt{consume}\ e' \ = \ \texttt{if}\ n - e' \geq 0\ \texttt{then}\ n := n - e'\ \texttt{else}\ \texttt{ReservationExceeded}$$

where n is the amount of resources reserved but not consumed. Soundness means that if the initial state satisfies $\texttt{scg}(\texttt{consume}\ e')(P, e)$, then no errors occur in `consume` e', and the final state satisfies (P, e). If we have n_0 resources initially, then after `consume` e', we have $n_0 - e'$. According to the above definition of `consume`, no

$$
\begin{array}{lcl}
\mathtt{scg}_{prog}(f_1 \ldots f_n c) & = & \mathtt{scg}_{fun}(f_1) \wedge \ldots \wedge \mathtt{scg}_{fun}(f_n) \wedge \mathtt{scg}_{cmd}(c) \\
\\
\mathtt{scg}_{cmd}(c) & = & \forall \vec{x}.\mathtt{true} \Rightarrow (P \wedge 0 \geq e) \\
 & & \text{where } (P, e) = \mathtt{scg}(c)(\mathtt{true}, 0) \\
 & & \text{and} \quad \vec{x} \text{ are the variables in } c \\
\\
\mathtt{scg}_{fun}(\mathtt{fun}\ f\ \mathtt{req}\ (A_0, e_0)\ \mathtt{ens}\ (A_1, e_1)\ c\ \mathtt{end}) & = & \forall \vec{x}.A_0 \Rightarrow (P \wedge e_0 \geq e) \\
 & & \text{where } (P, e) = \mathtt{scg}(c)(A_1, e_1) \\
 & & \text{and} \quad \vec{x} \text{ are the variables in } c \\
\\
\mathtt{scg}(\mathtt{skip})(P, e) & = & (P, e) \\
\mathtt{scg}(c_1; c_2)(P, e) & = & \mathtt{scg}(c_1)(\mathtt{scg}(c_2)(P, e)) \\
\mathtt{scg}(x := e')(P, e) & = & ([e'/x]P, [e'/x]e) \\
\mathtt{scg}(\mathtt{consume}\ e')(P, e) & = & (P, \mathtt{cond}(e \geq 0, e, 0) + e') \\
\mathtt{scg}(\mathtt{reserve}\ e')(P, e) & = & (P, \mathtt{cond}(e \geq 0, e, 0) - e') \\
\mathtt{scg}(\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2)(P, e) & = & (\mathtt{cond}(b, P_1, P_2), \mathtt{cond}(b, e_1, e_2)) \\
 & & \text{where } (P_1, e_1) = \mathtt{scg}(c_1)(P, e) \\
 & & \text{and} \quad (P_2, e_2) = \mathtt{scg}(c_2)(P, e) \\
\mathtt{scg}(\mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{inv}\ (A_I, e_I))(P, e) & = & (A_I \wedge \forall \vec{x}.A_I \Rightarrow \mathtt{cond}(b, Q', Q), e_I) \\
 & & \text{where } (P', e') = \mathtt{scg}(c)(A_I, e_I) \\
 & & \text{and} \quad Q' = P' \wedge e_I \geq e' \\
 & & \text{and} \quad Q = P \wedge e_I \geq e \\
 & & \text{and} \quad \vec{x} \text{ are the variables modified in } c \\
\mathtt{scg}(\mathtt{call}\ f)(P, e) & = & (A_0 \wedge \forall \vec{x}.A_1 \Rightarrow (P \wedge e_1 \geq e), e_0) \\
 & & \text{where } \mathtt{fun}\ f\ \mathtt{req}\ (A_0, e_0)\ \mathtt{ens}\ (A_1, e_1)\ c\ \mathtt{end} \\
 & & \text{and} \quad \vec{x} \text{ are the variables modified in } c
\end{array}
$$

Fig. 6. Definition of scg

$$
\begin{array}{l}
\forall N.\mathtt{true} \Rightarrow \\
\forall i.i \leq N \Rightarrow \\
\quad \mathtt{cond}(\ i < N, \\
\qquad\quad i + 1 \leq N \wedge N - i \geq \mathtt{cond}(N - i - 1 \geq 0, N - i - 1, 0) + 1, \\
\qquad\quad \mathtt{true} \wedge N - i \geq 0) \wedge \\
\quad 0 \geq \mathtt{cond}(N - 0 \geq 0, N - 0, 0) - N
\end{array}
$$

Fig. 7. scg of *Mixed1* from Figure 2

errors means that $n_0 - e' \geq 0$. And the final state satisfies (P, e) if $n_0 - e' \geq e$. So we have $n_0 - e' \geq \mathtt{cond}(e \geq 0, e, 0)$. The scg clause for consume e' places exactly this requirement on n_0.

As an example of the scg in practice, Figure 7 shows the verification condition for the program *Mixed1* of Figure 2.

5.1 Soundness

The soundness theorem is:

THEOREM 1. *For all* $\sigma, n, c, P, e, R, \sigma', n'$, *if*

$$\langle \sigma, n \rangle \models \mathtt{scg}(c)(P, e) \text{ and}$$
$$\langle c, \sigma, n \rangle \Downarrow R$$

then

$$R \neq \texttt{ReservationExceeded}\ \textit{and}$$
$$\textit{If}\ R = \langle \sigma', n' \rangle, \textit{then}\ R \models (P, e)$$

Proof: By induction on the command c. See the Appendix for details.

5.2 Annotator

As it stands, our approach requires the programmer manually to insert calls to `reserve`, write loop invariants, and add function pre and post conditions. We are currently working on an annotation tool that automatically and correctly adds these assertions, similar in spirit to Houdini [Flanagan and Leino 2001]. Although finding optimal annotations is undecidable, the tool can "fall back" to inserting a `reserve` before each `consume`. This annotation scheme is verifiable using the trivial loop invariant `true`, and it removes the need for hand annotation when the programmer does not care about efficiency. Beyond this "base line" performance, we plan to include a knowledge base of common loop idioms and their invariants.

One advantage of manual annotation is that the programmer can decide how early to reserve resources. It is less costly to reserve all resources at once, but it is also "anti-social" to hold unused resources, preventing other concurrently running programs from using them. The programmer can also decide whether to reserve exactly the right amount of resources, which may be difficult to determine, or whether to over-estimate.

6. PROVER

This section describes how to efficiently prove the safety conditions. We observe that the grammar for predicates restricts the left side of implications to annotations, not full predicates. Annotations are conjunctions of boolean expressions that are equalities or comparisons between integer expressions.

We also observe that the our definition of `scg` respects this restriction. In particular, all formulas on the left side of an implication arise from loop invariants and pre and post conditions.

Thus, we can use a simple theorem prover $\texttt{prove} : a \times p \rightarrow \mathit{Bool}$ where $\texttt{prove}(A, P)$ holds if and only if $A \Rightarrow P$ is valid. Valid means that the formula is true for all values of the global variables and fresh constants introduced by the rule for universal quantification. Figure 8 shows the definition of `prove`.

To prove $A \Rightarrow P$, `prove` recursively decomposes P until it reaches a boolean expression b. Note that it only decomposes the right side of the implication; it leaves the left side alone. It then uses a satisfiability procedure `sat` to check whether $A \Rightarrow b$ is valid. As usual, $A \Rightarrow b$ is valid if and only if its negation $A \wedge \neg b$ is unsatisfiable. Since the form of A is restricted, we only call `sat` on a conjunction of (possibly

$$\begin{array}{lcl}
\texttt{prove}(A, b) & = & \neg\texttt{sat}(A \wedge \neg b) \\
\texttt{prove}(A, P_1 \wedge P_2) & = & \texttt{prove}(A, P_1) \wedge \texttt{prove}(A, P_2) \\
\texttt{prove}(A, A_1 \Rightarrow P) & = & \texttt{prove}(A \wedge A_1, P) \\
\texttt{prove}(A, \forall x.P) & = & \texttt{prove}(A, [a/x]P) \quad (a \text{ is fresh}) \\
\texttt{prove}(A, \texttt{cond}(b, P_1, P_2)) & = & \texttt{prove}(A \wedge b, P_1) \wedge \texttt{prove}(A \wedge \neg b, P_2)
\end{array}$$

Fig. 8. Definition of `prove`

negated) boolean expressions. Since `prove` decomposes P using invertible rules, it is sound and complete if and only if `sat` is sound and complete.

We can implement `sat` by combining decision procedures for equality and linear arithmetic, as first described in [Nelson and Oppen 1979] and [Shostak 1984], and more recently in [Shankar and Ruess 2002]. In our experiments, we used ESC/Java [Flanagan et al. 2002] to generate and prove safety conditions from Java code. ESC/Java uses the Simplify theorem prover [Detlefs et al. 2003], which is sound and complete for our class of conditions.

Although we can probably trust our simple recursive prover, we may not want to trust the more complex satisfiability procedure at its core. To address this problem, we can use proof-carrying code [Necula 1997] and require the program producer to send a safety proof to the program consumer. If the satisfiability procedure generates verifiable proofs, then the producer can create a safety proof by running the `prove` procedure and collecting all the satisfiability proofs. The program consumer can check the proof by running the `prove` procedure, just as the producer did, and checking each of the satisfiability proofs. We may also choose to use PCC if we enrich the language of invariants and replace our simple prover with a more complex first-order prover.

7. `TAR` EXAMPLE

This section describes our experience with a version of `tar` written in Java. We wanted to see how hard it would be to annotate a "real" program, whether we could report policy violations earlier, and whether we could reduce the cost of dynamic checks. We chose a security policy with two quotas, one to limit the number of bytes *read* from the file system, and one to limit the number of bytes *written*. In this example, we leave memory management to Java's garbage collector. If we were verifying a C program, we would be more inclined to manage memory explicitly.

We began with a Java `tar` program from ICE Engineering [Endres 2003] but removed some features to simplify the annotation process. The final program has 1700 lines of code, of which 577 lines are relevant to I/O.

We prototyped our ideas using ESC/Java [Flanagan et al. 2002], which checks pre and post conditions for Java code using the Simplify theorem prover [Detlefs et al. 2003]. Using the implementation of `reserve` and `consume` shown in Figure 9, ESC/Java generates essentially the same verification condition as the `scg` function. Note that this code is trusted, while the code for `tar` itself is untrusted.

Although ESC/Java was excellent for prototyping our ideas, it is not suitable for verifying code safety. First, it is unsound, because it does not throughly check loop invariants and side-effect assertions (`modifies`). Thus, it cannot form the basis for a secure system. Second, it does not generate certificates for later verification. Third, it is too large to run on mobile devices. For these reasons, we are developing a lightweight implementation based on a certificate-generating prover [Necula and Rahul 2001].

The implementation of our ideas in ESC/Java is straightforward. We maintain two pools, a static pool and a dynamic pool. The static pool is the amount of resources reserved but not yet consumed. In the operational semantics, it is represented by n. In ESC/Java, we represent it using a *ghost variable* that exists only at verification time.

```
1  private static long dynamicPoolRead = 0;
2  //@ ghost public static long staticPoolRead = 0;
3  //@ invariant staticPoolRead >= 0;
4  //@ invariant dynamicPoolRead >= 0;
5
6  //@ requires n >= 0;
7  //@ ensures staticPoolRead == \old(staticPoolRead) + n;
8  //@ modifies dynamicPoolRead, staticPoolRead;
9  public static void reserveRead (long n) {
10   if (dynamicPoolRead >= n) {
11     dynamicPoolRead -= n;
12     //@ set staticPoolRead = staticPoolRead + n;
13   } else {
14     System.out.println ("Read quota exceeded!\n");
15     System.exit (1);
16   }
17 }
18
19 //@ requires n >= 0 && staticPoolRead >= n;
20 //@ ensures staticPoolRead == \old(staticPoolRead) - n;
21 //@ modifies staticPoolRead;
22 public static void consumeRead (long n) {
23   //@ set staticPoolRead = staticPoolRead - n;
24 }
```

Fig. 9. Trusted implementation of `reserve` and `consume` in ESC/Java.

The dynamic pool is difference between the program's resource quota, determined by the security policy, and the amount of resources the program has reserved. In the operational semantics, the dynamic pool was not explicitly represented, and the `ReserveFailed` error occured non-deterministically. In ESC/Java, we represent it using a standard runtime global variable.

At the start of execution, the runtime system sets the dynamic pool to the program's resource quota. The `reserve` operation transfers resources from the dynamic pool to the static pool. The `consume` operation removes resources from the static pool. The invariants ensure that neither pool drops below empty. Note that ESC/Java verifies each method's implementation against its specification using only the *specifications* of the methods that it calls, not their implementations.

The naïve `tar` implementation requires two dynamic checks for each 512-byte block, one for `read` and one for `write`. Using reservations, our implementation perform two checks per *file* rather than two checks per *block*. Figure 10 shows the code to write a file to the archive. For verification to succeed, we replaced the usual "while not EOF" loop with a `for` loop that counts a definite number of blocks.

Ideally, we would like to perform only two checks to create the entire archive. We haven't tried this experiment yet, but the code would need to prescan the directories to build a table of file sizes. The prover would need to connect the loop that sums the file sizes to the loop that reads the files.

We annotated each I/O method by computing its resource use in terms of the resource use of its subroutines. If a method's use was dynamic or difficult to state in closed form, we added a dynamic check to stop its upward propagation ("the buck stops here"). In general, the program can reserve more resources than it actually

```
1  long size = file.length ();
2  long q = size / recordSize;
3  long r = size % recordSize;
4  long size2 = q * recordSize;
5  long size3 = size2 + (r > 0) ?  recordSize :  0;
6
7  Resources.reserveWrite (size3 + recordSize);
8  Resources.reserveRead (size);
9  out.writeHeaderRecord (entry);
10
11 for (int i = 0; i < q; ++i) {
12   in.read (buffer, 0, recordSize);
13   out.writeRecord (buffer);
14 }
15
16 if (r > 0) {
17   Arrays.fill (buffer, (byte) 0);
18   in.read (buffer, 0, r);
19   out.writeRecord (buffer);
20 }
```

Fig. 10. Untrusted Java `tar` code excerpt.

uses, but we found it unnecessary to overestimate, since precise accounting was easy enough. In total, `tar` required 33 lines of annotation.

We tested `tar` on a directory containing 13.4 mb in 1169 files, for an average file size of 11.7 kb. The unannotated program performed 57210 I/O operations on 512-byte blocks. Since each operation requires a dynamic check, it also performed 57210 dynamic checks. The annotated program also performed 57210 I/O operations. However, since it performed one dynamic check per file rather than per block, it only performed 2389 dynamic checks. That is, it performed almost 24 times fewer dynamic checks. Of course, this ratio is the average file size divided by 512.

Because block I/O operations are much more expensive than dynamic checks, we did not obtain a corresponding decrease in overall run time. However, our technique also applies to operations where the check is expensive relative to the operation itself, such as instruction counting, where the limited resource is the number of instructions the program executes, memory reference counting, where the limited resource is the number of memory references the program makes, and array bounds checking, where the static verifier checks that all array references are in-bounds. Indeed, a simple experiment with `gcc` shows that removing an array bounds check from a tight loop reduces runtime by 33%.

8. RELATED WORK

Our work combines ideas from several areas: Dijkstra's weakest precondition computation [Dijkstra 1976], Necula and Lee's proof-carrying code [Necula 1997], partial evaluation's separation of static and dynamic binding times [Jones et al. 1993], and standard compiler optimizations such as hoisting and array bounds check elimination [Gupta 1993].

Since we *combine* static and dynamic checking, our work is only tangentially related to purely static approaches such as Resource Bound Certification [Crary

and Weirich 2000] and Mobile Resource Guarantees [Hofmann and Jost 2003] or purely dynamic approaches such as the Java security monitor [Gong 1999]. The implementations based on bytecode rewriting [Czajkowski and von Eicken 1998; Evans and Twyman 1999; Erlingsson and Schneider 1999; Colcombet and Fradet 2000; Pandey and Hashii 2000; Chander et al. 2001; Kim et al. 2001] are also purely dynamic, since they add checks without performing significant static analysis.

Our approach is a non-trivial instance of Necula and Lee's safe kernel extension method [Necula and Lee 1996]. They show that the OS designer can export an unsafe, low-level API if he provides a set of rules for its use, and a static analysis that checks whether clients follow these rules. By contrast, most designers wrap the low-level API in a safe but inefficient high-level API that clients can call without restriction. For array bounds checking, the low-level API is the unguarded reference, while the high-level API guards the reference with a bounds check. The usage rule is that the index must be in bounds.

In our case, the low-level API is `reserve` and `consume`. The high-level API, which we intentionally avoid, immediately prefixes `consume` by `reserve`, so that each `consume` has enough resources. This high-level API provides pure dynamic checking. The usage rule is that we `reserve` some time before we `consume`, but not necessarily immediately before. We extricate this useful, low-level API from its high-level wrapper and provide a flexible but safe set of usage rules, which we show how to statically check efficiently. The end result is a novel combination of static and dynamic checking.

On the surface, our work seems similar to approaches that place dynamic checks according to static analysis, such as Wallach's SAFKASI system [Wallach et al. 2000] and Gupta's elimination and hoisting of array bounds checks [Gupta 1993]. These systems limit the programmer to the safe, high-level API, but they inline and optimize calls to it according to the low-level API's usage rules and semantics. By contrast, like PCC, we separate verification from optimization, which is untrusted and can be performed by the programmer or by an automated tool. The programmer can also ignore the high-level API and call the low-level API directly.

The paper [Patel and Lepreau 2003] also describes a hybrid (mixed static and dynamic) approach to resource accounting. They use static analysis of execution time to reject some overly expensive active network router extensions. They use dynamic checks to monitor other, unspecified resources. At this level of detail, their static and dynamic checks are not tightly coupled. However, they also use static analysis to locate dynamic network polling operations. They bind their ideas closely to the complex active network setting and do not extract a simple, reuseable API or a proof system for reasoning about it.

The TALT-R system [Vanderwaart and Crary 2005] statically verifies that a program performs a `yield` operation after every Y instructions. Their language appears similar to ours, but yielding twice does not allow a program to execute $2Y$ instructions. Thus, their language does not allow the programmer to hoist and combine dynamic checks.

9. EXTENSIONS AND FUTURE WORK

We are currently engaged in future work in several different areas. First, due to the limitations of existing tools, we are developing an SCG and prover that can

prove resource-use safety for Java bytecode and produce proof witnesses. This effort presents several engineering challenges, such as scaling our SCG to a larger language, tracking source level annotations in bytecode, and building an efficient proof checker that performs well on mobile devices. Second, we are designing a tool that automatically and correctly annotates bytecode with resource reservations. Third, we are applying our techniques to other security mechanisms such as stack inspection and access control. Fourth, we are investigating situations where the check is expensive relative to the operation itself, such as instruction counting and memory reference sandboxing.

10. CONCLUSION

We have demonstrated a novel API for enforcing resource bounds that allows the programmer to trade-off intelligently between the static and dynamic approaches. In most of the program, the programmer places `reserve` immediately before `consume`. However, in inner loops, the programmer works hard to hoist and combine `reserve` operations. Thus, when intelligently applied, our approach offers the best of both worlds. Like dynamic checking, it handles complex programs and is relatively easy to apply. Like static checking, it is efficient and detects errors early.

By adapting ideas from weakest preconditions and proof-carrying code, we showed how the code consumer can statically verify that programs do not exceed their resource bounds. We presented a practical language and showed how to implement a sound but efficient verifier for it. Finally, we described our experience annotating and verifying a Java version of `tar` for resource safety.

Furthermore, our approach generalizes to APIs other than resource checking. At present, code consumers hide these APIs in high-level wrappers that are safe but inefficient. Using our hybrid approach, code consumers can give code producers direct access to efficient, low-level APIs without sacrificing safety.

APPENDIX

We show the proof of the soundness theorem for the safety condition generator `scg`.

THEOREM 1. *For all* $\sigma, n, c, P, e, R, \sigma', n'$*, if*

$$\langle \sigma, n \rangle \models \texttt{scg}(c)(P, e) \text{ and}$$
$$\langle c, \sigma, n \rangle \Downarrow R$$

then

$$R \neq \texttt{ReservationExceeded} \text{ and}$$
$$\text{If } R = \langle \sigma', n' \rangle, \text{then } R \models (P, e)$$

Proof: By induction on the command c. We show the cases for `consume` and `reserve`. For `consume`, we want

$$\langle \sigma, n \rangle \models \texttt{scg}(\texttt{consume } e')(P, e) \wedge$$
$$\langle \texttt{consume } e', \sigma, n \rangle \Downarrow R$$
$$\Rightarrow$$
$$R \neq \texttt{ReservationExceeded} \wedge$$
$$R = \langle \sigma', n' \rangle \Rightarrow R \models (P, e)$$

By the definition of `scg`, we want

$$\begin{array}{l}\langle\sigma, n\rangle \models (P, \texttt{cond}(e \geq 0, e, 0) + e') \wedge \\ \langle \texttt{consume}\ e', \sigma, n\rangle \Downarrow R \\ \Rightarrow \\ R \neq \texttt{ReservationExceeded} \wedge \\ R = \langle\sigma', n'\rangle \Rightarrow R \models (P, e)\end{array}$$

Because $\langle\sigma, n\rangle \models e$, only rule C-OK applies, so we want

$$\begin{array}{l}\langle\sigma, n\rangle \models (P, \texttt{cond}(e \geq 0, e, 0) + e') \\ \Rightarrow \\ \langle\sigma, n - [\![e']\!]\sigma\rangle \models (P, e)\end{array}$$

By the definition of $\models$, we want

$$\begin{array}{l}n \geq [\![\texttt{cond}(e \geq 0, e, 0) + e']\!]\sigma \\ \Rightarrow \\ n - [\![e']\!]\sigma \geq [\![e]\!]\sigma\end{array}$$

which holds by arithmetic. For `reserve`, we want

$$\begin{array}{l}\langle\sigma, n\rangle \models \texttt{scg}(\texttt{reserve}\ e')(P, e) \wedge \\ \langle \texttt{reserve}\ e', \sigma, n\rangle \Downarrow R \\ \Rightarrow \\ R \neq \texttt{ReservationExceeded} \wedge \\ R = \langle\sigma', n'\rangle \Rightarrow R \models (P, e)\end{array}$$

By the definition of `scg`, we want

$$\begin{array}{l}\langle\sigma, n\rangle \models (P, \texttt{cond}(e \geq 0, e, 0) - e') \wedge \\ \langle \texttt{reserve}\ e', \sigma, n\rangle \Downarrow R \\ \Rightarrow \\ R \neq \texttt{ReservationExceeded} \wedge \\ R = \langle\sigma', n'\rangle \Rightarrow R \models (P, e)\end{array}$$

There are three rules for `reserve`, R-OK, R-FAIL1, and R-FAIL2. Rule R-FAIL1 does not apply because $\langle\sigma, n\rangle \models -e$. If rule R-FAIL2 applies, we want

$$\texttt{ReserveFailed} \neq \texttt{ReservationExceeded}$$

which holds trivially. If rule R-OK applies, we want

$$\begin{array}{l}\langle\sigma, n\rangle \models (P, \texttt{cond}(e \geq 0, e, 0) - e') \\ \Rightarrow \\ \langle\sigma, n + [\![e']\!]\sigma\rangle \models (P, e)\end{array}$$

By definition of $\models$, we want

$$\begin{array}{l}n \geq [\![\texttt{cond}(e \geq 0, e, 0) - e']\!]\sigma \\ \Rightarrow \\ n + [\![e']\!]\sigma \geq [\![e]\!]\sigma\end{array}$$

which holds by arithmetic. □

REFERENCES

Chander, A., Espinosa, D., Islam, N., Lee, P., and Necula, G. 2005. JVer: a Java verifier. In *Computer Aided Verification*. Edinburgh, Scotland.

Chander, A., Mitchell, J., and Shin, I. 2001. Mobile code security by Java bytecode instrumentation. In *DARPA Information Survivability Confernce and Exposition*.

Colcombet, T. and Fradet, P. 2000. Enforcing trace properties by program transformation. In *Principles of Programming Languages*. Boston, Massachusetts.

Crary, K. and Weirich, S. 2000. Resource bound certification. In *Principles of Programming Languages*. Boston, Massachusetts.

Czajkowski, G. and von Eicken, T. 1998. JRes: a resource accounting interface for Java. In *Object-Oriented Programming, Systems, Languages, and Applications*. Vancouver, BC.

Detlefs, D., Nelson, G., and Saxe, J. 2003. Simplify: a theorem prover for program checking. Tech. Rep. HPL-2003-148, HP Laboratories. July.

Dijkstra, E. 1976. *A Discipline of Programming*. Prentice-Hall.

Endres, T. 2003. Java Tar 2.5. `http://www.trustice.com`.

Erlingsson, U. and Schneider, F. 1999. SASI enforcement of security policies: a retrospective. In *New Security Paradigms Workshop*. Caledon, Canada.

Evans, D. and Twyman, A. 1999. Flexible policy-directed code safety. In *Security and Privacy*. Oakland, California.

Flanagan, C. and Leino, K. R. M. 2001. Houdini, an annotation assistant for ESC/Java. In *Formal Methods Europe (LNCS 2021)*. Berlin, Germany.

Flanagan, C., Leino, R., Lilibridge, M., Nelson, G., Saxe, J., and Stata, R. 2002. Extended static checking for Java. In *Programming Language Design and Implementation*. Berlin, Germany.

Gong, L. 1999. *Inside Java 2 Platform Security*. Addison-Wesley.

Gupta, R. 1993. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems 2,* 1-4 (March–December), 135–150.

Hofmann, M. and Jost, S. 2003. Static prediction of heap space usage for first-order functional programs. In *Principles of Programming Languages*. New Orleans, Louisiana.

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

Kim, M., Kannan, S., Lee, I., and Sokolsky, O. 2001. Java-MaC: a run-time assurance tool for Java programs. *Electronic Notes in Theoretical Computer Science 55,* 2.

Mitchell, J. C. 1996. *Foundations for Programming Languages*. MIT Press.

Necula, G. 1997. Proof-carrying code. In *Principles of Programming Languages*. Paris, France.

Necula, G. and Lee, P. 1996. Safe kernel extensions without run-time checking. In *Operating Systems Design and Implementation*. Seattle, Washington.

Necula, G. C. and Rahul, S. P. 2001. Oracle-based checking of untrusted software. In *Principles of Programming Languages*. London, England.

Nelson, G. and Oppen, D. 1979. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems 1,* 2 (Oct.), 245–257.

Pandey, R. and Hashii, B. 2000. Providing fine-grained access control for Java programs via binary editing. *Concurrency: Practice and Experience 12*, 1405–1430.

Patel, P. and Lepreau, J. 2003. Hybrid resource control of active extensions. In *Open Architectures and Network Programming*. San Francisco, California.

Shankar, N. and Ruess, H. 2002. Combining Shostak theories. In *Rewriting Techniques and Applications*. Copenhagen, Denmark.

Shostak, R. E. 1984. Deciding combinations of theories. *Journal of the ACM 31,* 1 (Jan.), 1–12.

Vanderwaart, J. and Crary, K. 2005. Automated and certified conformance to responsiveness policies. In *Workshop on Types in Language Design and Implementation*. Long Beach, California.

Wallach, D., Appel, A., and Felten, E. 2000. SAFKASI: a security mechanism for language-based systems. *Transactions on Software Engineering 9,* 4 (October), 341–378.

Received September 2005; accepted ???