# Enforcing Resource Bounds via Static Verification of Dynamic Checks

Ajay Chander[1], David Espinosa[1], Nayeem Islam[1], Peter Lee[2], and George Necula[3]

[1] DoCoMo Labs USA, San Jose, CA, fax 408-573-1090
chander,espinosa,islam@docomolabs-usa.com
[2] Carnegie Mellon University, Pittsburgh, PA
Peter.Lee@cs.cmu.edu
[3] University of California, Berkeley, CA
necula@eecs.berkeley.edu

**Abstract.** We classify existing approaches to resource-bounds checking as static or dynamic. Dynamic checking performs checks *during* program execution, while static checking performs them *before* execution. Dynamic checking is easy to implement but incurs runtime cost. Static checking avoids runtime overhead but typically involves difficult, often incomplete program analyses. In particular, static checking is hard in the presence of dynamic data and complex program structure. We propose a new resource management paradigm that offers the best of both worlds. We present language constructs that let the code producer optimize dynamic checks by placing them either before each resource use, or at the start of the program, or anywhere in between. We show how the code consumer can then statically verify that the optimized dynamic checks enforce his resource bounds policy. We present a practical language that is designed to admit decidable yet efficient verification and prove that our procedure is sound and optimal. We describe our experience verifying a Java implementation of `tar` for resource safety. Finally, we outline how our method can improve the checking of other dynamic properties.

## 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 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 usage bounds 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 performs significantly fewer runtime checks while enforcing resource bounds than previous methods.

We address the scenario in which a *code consumer* runs an *untrusted* program created by a *code producer*, who possibly is untrusted. This program communicates with the code consumer's computer via a 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 resources according to the code consumer's *security policy*. This policy specifies the resources that each program can use, along with the corresponding usage bounds.

Our technique enforces resource usage bounds with a combination of static and dynamic checks. More precisely, we verify statically that a program's dynamic checks are sufficient to enforce the consumer's safety policy. In order to support such hybrid checking, we separate the `acquire` function, which *acquires* a resource, from the various functions that *consume* the resource. For notational simplicity, we use a single, specific `consume` function to represent abstractly any library function that consumes resources.

In current libraries, `acquire` and `consume` are performed together when a resource is used. It is easy to automatically replace these calls by pairs of separate calls to `acquire` and `consume`. It is also easy to verify statically that the result of this transformation never uses more resources than have been acquired.

The advantage of this separation is that the programmer, or appropriate optimization tools, can combine multiple `acquires` into one and can hoist `acquire` out of a loop whose body consumes resources. In this paper, we describe a static analysis that verifies that an arbitrary placement of `acquires` is sufficient. The analysis is decidable and efficient, and our experiments show that it can validate even aggressive optimizations. Moving `acquire` out of a loop can yield an arbitrary improvement in in the number of dynamic checks. This improvement results in significant performance gains if the `acquire` operation consults a complex or remote resource manager. Moving checks earlier can also guarantee that no resource errors occur in critical code fragments such as atomic transactions.

We begin this paper by introducing an imperative language with resource-aware constructs in Section 2, and illustrate the benefits of our method over purely static or dynamic approaches using a few key examples. In Section 3, we present an operational semantics for our language, and provide a precise characterization of resource-use safety. Section 4 describes the two components of the verifier: the safety condition generator (SCG) (Section 4.1) and the prover (Section 4.4), and presents soundness and optimality results for our SCG. We describe our experience with the `tar` program in Section 5. Section 6 positions this paper with respect to relevant work in a few areas. We mention ongoing efforts and future work opportunities in Section 7, and conclude in Section 8.

## 2 Concept

For resource-usage safety, we must ensure that each resource consuming operation, denoted by `consume`, has adequate resources available, as specified by a system security policy. Abstractly, we can refer to this policy as `quota`, and so the sum of all of the `consumes` must be guaranteed never to exceed `quota`; we state this goal informally as `consume` $\leq$ `quota`. In the following, we motivate our approach to the resource-usage problem with a few examples, and introduce our method over a simple iterative language.

### 2.1 Examples

Figure 1 shows four programs that use resources. Program *Dynamic* uses an amount of resources that depends entirely on its runtime input. Program *Static* uses a fixed amount of resources. Program *Mixed1* uses a fixed amount of resources, but this amount is dynamic. Program *Mixed2* uses a fixed amount of resources each time through its inner loop, but it executes this loop a dynamic number of times.

A standard dynamic checker 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.

```
Program Dynamic
  while read() $\neq 0$
    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() $\neq 0$
    $i$ := $0$
    while $i < 100$
      consume $1$
      $i$ := $i+1$
```

**Fig. 1.** Example programs

### 2.2 Language

In order to describe the static checking procedure, we use a simple imperative programming language that computes with integer values. Without loss of generality, we assume that there is one resource of interest whose amount is measured in some arbitrary unit. We introduce the command `consume` $e$ to model any operation that uses $e$ units of the resource, where $e$ is an expression in the language. We introduce the command `acquire` $e$ to reserve $e$ resource units from the operating system. This command may fail, but if it succeeds, we know that $e$ resource units have been reserved for the running program. The `acquire` operation is an example of a dynamic *reservation* instruction, perhaps realized with a library function, and occurs only in programs created by the code producer. In contrast, the `consume` operation acts as a no-op at execution time, and is used only in the static verification process.

Figure 2 shows the syntax of the full language. We assume that the variables $x$ take only integer values. The expression $\mathtt{cond}(b, e_1, e_2)$ has value $e_1$ if the boolean expression $b$ has value `true` and has value $e_2$ otherwise. Similarly, the command cond($b$, $P_1$, $P_2$) is equivalent to the command $P_1$ if $b$ has value `true`, and command $P_2$ otherwise. The propositional connectives $\wedge, \forall, \Rightarrow$ have their usual meaning.

The argument $e$ to `acquire` and `consume` must be non-negative. The safety condition generator of Section 4.1 statically guarantees this condition.

Note also that we annotate the looping command with an invariant $(A, e)$. During static checking, we verify that the predicate $A$ holds and there are at least $e$ resource units available before the looping command is executed. To simplify the task of the static checker, and to allow for a prover that is complete over safety conditions generated from programs in this language, we restrict the invariants to a conjunction of boolean equalities and comparisons between integer expressions and we similarly restrict the left side of implications in predicates.

### 2.3 Annotated Examples

The programmer's (or automated tool's) job is to insert enough `acquire` operations to make the program safe. It is always possible to insert an `acquire` before each `consume`, so that each `consume` performs a runtime check, bringing us to the pure dynamic checking safety paradigm. The question is whether the programmer or automated tool can insert fewer `acquire` operations and thereby reduce the cost of dynamic checking.

Figure 3 shows the same four programs with `acquire` operations added. Note that all four programs execute safely. *Dynamic* performs exactly the same checks that it

| | | |
|---|---|---|
| $e \ ::=$ | $x \mid n \mid e_1 + e_2 \mid n * e \mid \mathtt{cond}(b, e_1, e_2)$ | Integer expressions |
| $b \ ::=$ | $\mathtt{true} \mid e_1 \geq e_2 \mid e_1 = e_2$ | Boolean expressions |
| $c \ ::=$ | $\mathtt{skip} \mid x := e \mid c_1; c_2 \mid$ | |
| | $\mathtt{consume}\ e \mid \mathtt{acquire}\ e$ | |
| | $\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid$ | |
| | $\mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{inv}\ (A, e)$ | Commands |
| $P ::=$ | $b \mid P_1 \wedge P_2 \mid A \Rightarrow P \mid$ | |
| | $\forall x.P \mid \mathtt{cond}(b, P_1, P_2)$ | Predicates |
| $A ::=$ | $b \mid A_1 \wedge A_2$ | Annotations |

**Fig. 2.** Simple imperative language definition.

would in a dynamic system, 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, reserving all resources either at the beginning or each time through the outer loop; for example, *Mixed2* performs two orders of magnitude fewer checks.

```
Program Dynamic
  while read() $\neq 0$
    acquire $1$
    consume $1$
  inv (true, $0$)
```

```
Program Static
  acquire $10000$
  $i$ := $0$
  while $i < 10000$
    consume $1$
    $i$ := $i + 1$
  inv ($i \leq 10000$, $10000 - i$)
```

```
Program Mixed1
  $N$ := read()
  acquire $N$
  $i$ := $0$
  while $i < N$
    consume $1$
    $i$ := $i + 1$
  inv ($i \leq N$, $N - i$)
```

```
Program Mixed2
  while read() $\neq 0$
    acquire $100$
    $i$ := $0$
    while $i < 100$
      consume $1$
      $i$ := $i$ + 1
    inv ($i \leq 100$, $100 - i$)
```

**Fig. 3.** Example programs with annotations.

Notice that the new language abstractions provide us with a midpoint in the original resource-usage condition `consume` $\leq$ `quota`. That is, we check statically that `consume` $\leq$ `acquire`, and we check dynamically that `acquire` $\leq$ `quota`. Figure 4 illustrates this concept. Static checking lets us hoist and combine `acquire`s, so that we can use dynamically fewer of them and thus reduce the cost of checking.

## 3 Semantics of Annotated Programs

In this section, we formalize the meaning of expressions and commands, and make explicit the precise ways in which execution can fail, following well-known approaches to operational specifications of programming language constructs [1].

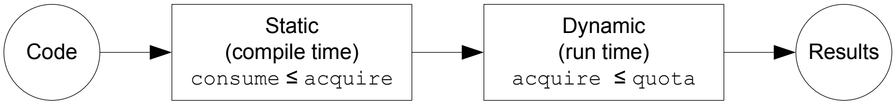

**Fig. 4.** Partitioning code safety into static and dynamic components.

The execution state is a pair $\langle \sigma, n \rangle$ of an environment $\sigma$ that maps variable names to integer values and a natural number $n$ that represents the amount of available resources. We write $\llbracket e \rrbracket \sigma$ for the value of the integer expression $e$ in the environment $\sigma$ and $\llbracket b \rrbracket \sigma$ for the value of the boolean expression $b$ in the environment $\sigma$. For example,

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

The other cases of the definition are straightforward. We use the notation $\sigma[x := n]$ to denote the environment that is identical to $\sigma$ except that $x$ is set to $n$.

### 3.1 Operational semantics

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 $R$. If there does not exist an $R$ such that $\langle c, \sigma, n \rangle \Downarrow R$, we write $\langle c, \sigma, n \rangle \Uparrow$ (pronounced "diverges").

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 an acquire fails, then $R$ is the error `AcquireFailed`. If the program uses more resources than it has acquired, then $R$ is the error `QuotaExceeded`. If the program does not satisfy an invariant annotation, then $R$ is the error `InvFailure`. Thus, from an initial state, a command either diverges, terminates normally, or terminates with one of three errors.

Figure 5 shows the operational semantics for while loops; the operational semantics for the other standard constructs is straightforward.

$$\frac{\sigma \not\models P}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (P, e), \sigma, n \rangle \Downarrow \texttt{InvFailure}}\ \textsc{WhileInvFailure}$$

$$\frac{\llbracket b \rrbracket \sigma = \texttt{false} \quad \sigma \models P \quad n \geq \llbracket e \rrbracket \sigma}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (P, e), \sigma, n \rangle \Downarrow \langle \sigma, n \rangle}\ \textsc{WhileF}$$

$$\frac{\begin{array}{l} \sigma \models P \;\; n \geq \llbracket e \rrbracket \sigma \;\; \langle c, \sigma, n \rangle \Downarrow \langle \sigma', n' \rangle \quad \llbracket b \rrbracket \sigma = \texttt{true} \;\; \sigma' \models P \;\; n' \geq \llbracket e \rrbracket \sigma' \\ \langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (P, e), \sigma', n' \rangle \Downarrow R \end{array}}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (P, e), \sigma, n \rangle \Downarrow R}\ \textsc{WhileT}$$

**Fig. 5.** Operational semantics for while loops.

Figure 6 shows the rules for evaluating resource-specific commands, which modify the amount of resources in the current state. Notice that only `acquire` replenishes this state, so that if the program starts with no resources, it must `acquire` all the resources that it uses. If enough resources are available, `consume` terminates normally, consuming resources. If not enough resources are available, it yields a `QuotaExceeded` error. The `acquire` command either increases the amount of available resources or yields an `AcquireFailed` error. In this formalization, the `acquire` command is non-deterministic. In practice, its behavior is determined by the operating system, which we do not model here. Alternatively, we could add an explicit dynamic pool to model the resources available to `acquire`.

$$\frac{n \geq [\![e]\!]\sigma}{\langle \texttt{consume}e, \sigma, n\rangle \Downarrow \langle \sigma, n - [\![e]\!]\sigma\rangle}\ \text{C-OK} \qquad \frac{n < [\![e]\!]\sigma}{\langle \texttt{consume}e, \sigma, n\rangle \Downarrow \texttt{QuotaExceeded}}\ \text{C-FAIL}$$

$$\frac{}{\langle \texttt{acquire}e, \sigma, n\rangle \Downarrow \langle \sigma, n + [\![e]\!]\sigma\rangle}\ \text{A-OK} \qquad \frac{}{\langle \texttt{acquire}e, \sigma, n\rangle \Downarrow \texttt{AcquireFailed}}\ \text{A-FAIL}$$

**Fig. 6.** Operational semantics for reservations

## 4 Verifier

The verifier has two parts, the safety condition generator (SCG), which computes a program's safety condition (SC), and the prover, which actually proves the SC. We define safety, state a soundness theorem, which says that the SC guarantees safety, and state an optimality theorem, which says that the SC captures *all* programs that are safe.

### 4.1 Safety condition generator

Our verifier uses a variant of Dijkstra's weakest precondition calculus [2]. We work with "generalized predicates" $(P, e)$, meaning that $P$ holds and there are at least $e$ resource units available. Figure 7 shows the definition of the safety condition generator `scg`. We define `scg` by recursion on the syntax of commands. Our definition matches the standard `scg` definition for all commands that do not manipulate resources explicitly. For the commands that manipulate resources, we extracted the definition from the soundness proof. The `scg` definition also (1) checks the invariant that there are a non-negative amount of resources available and (2) checks that the arguments to `acquire` and `consume` are non-negative.

Although our language uses structured control (`while` loops), we can also define `scg` for unstructured control (`goto`s), by associating an invariant with each label, or at least those at the heads of loops, as determined by a standard dominator-based control flow analysis.

### 4.2 Soundness

We write $\sigma \models P$ to indicate that predicate $P$ holds in state $\sigma$. Recall that $\sigma$ supplies values for the variables in $P$, so we define $\sigma \models P$ as usual by induction over the propositional connectives.

**Definition 1** $\langle \sigma, n\rangle \models (P, e)$ *iff* $n \geq 0$, $\sigma \models P$, *and* $\sigma \models n \geq e$.

$$\begin{aligned}
&\texttt{scg}(\texttt{skip})(P, e) && = (P, e)\\
&\texttt{scg}(c_1; c_2)(P, e) && = \texttt{scg}(c_1)(\texttt{scg}(c_2)(P, e))\\
&\texttt{scg}(x := e')(P, e) && = ([e'/x]P, [e'/x]e)\\
&\texttt{scg}(\mathsf{consume} e')(P, e) && = (P \wedge e' \geq 0, e' + \texttt{cond}(e \geq 0, e, 0)\\
&\texttt{scg}(\texttt{acquire}\ e')(P, e) && = (P \wedge e' \geq 0, e - e')\\
&\texttt{scg}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)(P, e) && = (\texttt{cond}(b, P_1, P_2), \texttt{cond}(b, e_1, e_2))\\
& && \quad \text{where } (P_1, e_1) = \texttt{scg}(c_1)(P, e)\\
& && \quad \text{and} \quad (P_2, e_2) = \texttt{scg}(c_2)(P, e)\\
&\texttt{scg}(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (A_I, e_I))(P, e) && = (A_I \wedge \forall \boldsymbol{x}.A_I \Rightarrow \texttt{cond}(b, Q', Q), e_I)\\
& && \quad \text{where } (P', e') = \texttt{scg}(c)(A_I, e_I)\\
& && \quad \text{and} \quad Q' = P' \wedge e_I \geq e'\\
& && \quad \text{and} \quad Q = P \wedge e_I \geq e\\
& && \quad \text{and} \quad \boldsymbol{x} \text{ are the variables modified in } c
\end{aligned}$$

**Fig. 7.** Definition of `scg`

That is, $n$ is non-negative, $P$ holds in $\sigma$, and at least $e$ resources are available in $\sigma$.

**Definition 2** *A tuple* $(c, \langle \sigma, n \rangle, (P, e))$ *is* safe *iff one of the following holds:*

1. $\langle c, \sigma, n \rangle \Uparrow$*, or*
2. $\langle c, \sigma, n \rangle \Downarrow$ `AcquireFailed`*, or*
3. $\langle c, \sigma, n \rangle \Downarrow \langle \sigma', n' \rangle$ *where* $\langle \sigma', n' \rangle \models (P, e)$*.*

That is, the `InvFailure` and `QuotaExceeded` errors do not occur, and $(P, e)$ holds if execution terminates normally.

**Definition 3** *A tuple* $(c, (P_0, e_0), (P, e))$ *is* safe *iff* $(c, \langle \sigma, n \rangle, (P, e))$ *is safe for all states* $\langle \sigma, n \rangle$ *such that* $\langle \sigma, n \rangle \models (P_0, e_0)$*.*

That is, $(P_0, e_0)$ guarantees safety. We can now state the soundness theorem:

**Theorem 1.** *For all* $c, P, e$*,* $(c, \texttt{scg}(c)(P, e), (P, e))$ *is safe.*

*Proof:* By structural induction on the derivation $\langle c, \sigma, n \rangle \Downarrow R$.

To check whether a given command is safe to execute, we check whether $\langle \sigma_0, 0 \rangle \models \texttt{scg}(c)(\texttt{true}, 0)$. That is, we compute the command's safety condition and check whether it holds in the initial execution state $\sigma_0$. If it holds, then $c$ cannot produce the `InvFailure` or `QuotaExceeded` errors. Thus, we do not check for them dynamically in actual practice, and we do not maintain the static resource pool $n$. Note that we *do* still check for dynamic policy violations, which raise the `AcquireFailed` error.

### 4.3 Optimality

The soundness theorem shows that our `scg` prevents $c$ from raising the `InvFailure` or `QuotaExceeded` errors. The `scg` is also optimal, meaning that it generates the weakest such condition, in the sense of:

**Definition 4** $(P_0, e_0) \Rrightarrow (P_1, e_1)$ *iff* $P_0 \Rightarrow (P_1 \wedge e_0 \geq e_1)$.

That is, whenever $(P_0, e_0)$ holds, so does $(P_1, e_1)$. The optimality theorem states:

**Theorem 2.** *For all* $c, P_0, e_0, P, e$*, if* $(c, (P_0, e_0), (P, e))$ *is safe, then* $(P_0, e_0) \Rrightarrow \texttt{scg}(c)(P, e)$*.*

*Proof:* By structural induction on the command $c$.

That is, whenever $(P_0, e_0)$ guarantees safety, $(P_0, e_0)$ implies $\texttt{scg}(c)$ $(P, e)$. Thus, $\texttt{scg}(c)$ $(P, e)$ is the weakest such condition.

### 4.4 Prover

In this section, we show how to 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 use a simple theorem prover `prove` : $a \times p \rightarrow Bool$ 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. A predicate $P$ is valid if and only if the $\texttt{prove}(\texttt{true}, P)$ is true. Figure 8 shows the definition of `prove`.

To prove $A \Rightarrow P$, `prove` recursively decomposes $P$ until it reaches a boolean expression $b$. 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 negated) boolean expressions. Since `prove` decomposes $P$ using invertible rules, it is sound and complete if and only if `sat` is sound and complete.

There are two notable satisfiability procedures that handle the linear inequalities that `scg` generates. One is due to Nelson and Oppen [3] and implemented by the Simplify prover [4] used in ESC/Java [5]. The other is due to Shostak [6] and implemented in PVS [7]. In our experiments, we used ESC/Java and Simplify to generate and prove safety conditions from Java code. For our class of conditions, Simplify is sound and complete.

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

**Fig. 8.** Definition of `prove`

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 [8] 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 if we enrich the language of invariants and replace our simple prover with a more complex first-order prover.

### 4.5 Annotator

As it stands, our approach requires the programmer manually to insert `acquire`s, write loop invariants, and add function pre and post conditions. We are currently working on a tool that automatically and correctly adds these assertions, similar in spirit to Houdini [9]. Although optimal annotation is undecidable, the tool can "fall back" to inserting an `acquire` 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 optimal annotations.

One advantage of manual annotation is that the programmer can decide how early to acquire resources. It is less costly to acquire 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 acquire exactly the right amount of resources, which may be difficult to determine, or whether to over-estimate.

### 4.6 Renewable Resources

We can easily extend our framework to handle renewable resources, such as memory and file handles, by allowing `acquire` and `consume` to take negative arguments. In essence, `acquire` and `consume` manage two pools, a *static pool* and a *dynamic pool*. With a positive argument, `acquire` moves resources from the dynamic pool to the static pool. With a positive argument, `consume` moves resources out of the static pool. With negative arguments, `acquire` and `consume` transfer resources in the opposite direction. The `consume` operation is part of the TCB in that it represents (or annotates) trusted library functions such as `malloc` (with positive argument) and `free` (with negative argument). The `acquire` operation is untrusted, and downloaded code is free to call it to obtain resources from the run-time system, or to release them back to the run-time system.

## 5 `Tar` example

In this section, we describe 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 that limits the number of bytes that `tar` reads and writes to the file system.

We began with a Java `tar` program from ICE Engineering [10] but revised it to simplify the annotation process. Although `tar` contains 1700 lines of code, we only needed to examine the 577 lines relevant to I/O.

We prototyped our ideas using ESC/Java [5], which checks pre and post conditions for Java code using the Simplify theorem prover [4]. Using the definitions of `acquire` and `consume` shown in Figure 9, ESC/Java generates essentially the same verification condition as the function shown in Figure 7. Although ESC/Java has been 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 [11].

The implementation of our ideas in ESC/Java is straightforward. We maintain two pools, a static pool and a dynamic pool. We represent the static pool using a *ghost variable* that exists only at verification time. At the start of execution, the user's security policy fills the dynamic pool with the program's resource quota. The `acquire` operation transfers resources from the dynamic pool to the static pool. The `consume` operation removes resources from the static pool. The invariants ensure that the pools never drop below empty. Note that ESC/Java verifies each method's implementation against its specification using only the specifications, not the implementations, of the methods that it calls.

The naive `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. We replaced the usual "while not EOF" loop with a `for` loop that counts a definite number of blocks. This structure ensures that `tar` does not exceed its quota even if a concurrent process lengthens the file.

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"). Although we experimented with annotations that overestimate resource use, we found that precise annotations were simple 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

```
private static long dynamicPoolRead = 0;
//@ ghost public static long staticPoolRead = 0;
//@ invariant staticPoolRead >= 0;
//@ invariant dynamicPoolRead >= 0;

//@ requires n >= 0;
//@ ensures staticPoolRead == \old(staticPoolRead) + n;
//@ modifies dynamicPoolRead, staticPoolRead;
public static void acquireRead (long n) {
  if (dynamicPoolRead >= n) {
    dynamicPoolRead -= n;
    //@ set staticPoolRead = staticPoolRead + n;
  } else {
    System.out.println ("Read quota exceeded!\n");
    System.exit (1);
  }
}

//@ requires n >= 0 && staticPoolRead >= n;
//@ ensures staticPoolRead == \old(staticPoolRead) - n;
//@ modifies staticPoolRead;
public static void consumeRead (long n) {
  //@ set staticPoolRead = staticPoolRead - n;
}
```

**Fig. 9.** Implementation of `acquire` and `consume` in ESC/Java.

```
long size = file.length ();
long q = size / recordSize;
long r = size % recordSize;
long size2 = q * recordSize;
long size3 = size2 + (r > 0) ? recordSize : 0;

Resources.acquireWrite (size3 + recordSize);
Resources.acquireRead (size);
out.writeHeaderRecord (entry);

for (int i = 0; i < q; ++i) {
  in.read (buffer, 0, recordSize);
  out.writeRecord (buffer);
}

if (r > 0) {
  Arrays.fill (buffer, (byte) 0);
  in.read (buffer, 0, r);
  out.writeRecord (buffer);
}
```

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

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 and memory reference counting.

## 6 Related work

Our work combines ideas from several areas: Dijkstra's weakest precondition computation [2], Necula and Lee's proof-carrying code [8], partial evaluation's separation of static and dynamic binding times [12], and standard compiler optimizations such as hoisting and array bounds check elimination [13].

Since we *combine* static and dynamic checking, our work is only tangentially related to purely static approaches such as Crary and Weirich's resource bound certification [14] or purely dynamic approaches such as the Java security monitor [15]. The implementations based on bytecode rewriting [16–22] 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 [23]. 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 `acquire` and `consume`. The high-level API, which we intentionally avoid, immediately prefixes `consume` by `acquire`, so that each `consume` has enough resources. This high-level API provides pure dynamic checking. The usage rule is that we `acquire` 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 [24] and Gupta's elimination and hoisting of array bounds checks [13]. 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.

Like us, Patel and Lepreau [25] describe 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.

Independently of us, Vanderwaart and Crary proposed the TALT-R system [26]. They place a *yield* at most every $Y$ instructions. That is, *yield* is similar to `acquire`$(Y)$. However, since *yield* does not itself debit a resource quota, it does not enable the fine-grained combination of static and dynamic checking.

## 7 Extensions and Future Work

Our approach can already handle (1) function pre and post conditions and (2) reuseable resources such as memory, but we do not have space to describe these extensions here.

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.

## 8 Conclusion

We have demonstrated a novel API for resource bounds enforcement that combines the best of static and dynamic approaches, by providing the means to dynamically reserve resources within programs and statically verify that the reservations are sufficient. Our soundness theorem gives the code consumer total confidence that statically verified programs do not exceed the resource bounds specified in his safety policy. Our approach gives the code producer (programmer or automated tool) complete freedom to optimize the placement of dynamic checks. Thus, we provide a system for writing statically verifiable resource-safe programs that handles dynamic data and complex program structure.

By adapting ideas from weakest preconditions and proof-carrying code, we showed how the code consumer can statically verify that resource reservations enforce his resource bounds policy. We presented a practical language that was carefully designed to admit decidable yet efficient verification and proved soundness and optimality theorems. Finally, we described our experience in successfully 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.

## References

1. Mitchell, J.C.: Foundations for Programming Languages. MIT Press (1996)
2. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
3. Nelson, G., Oppen, D.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems **1** (1979) 245–257
4. Detlefs, D., Nelson, G., Saxe, J.: Simplify: a theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories (2003)
5. Flanagan, C., Leino, R., Lilibridge, M., Nelson, G., Saxe, J., Stata, R.: Extended static checking for Java. In: Programming Language Design and Implementation, Berlin, Germany (2002)
6. Shostak, R.E.: Deciding combinations of theories. Journal of the ACM **31** (1984) 1–12
7. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In Kapur, D., ed.: 11th International Conference on Automated Deduction (CADE). Volume 607 of Lecture Notes in Artificial Intelligence., Saratoga, NY, Springer-Verlag (1992) 748–752
8. Necula, G.: Proof-carrying code. In: Principles of Programming Languages, Paris, France (1997)
9. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for esc/java. In: Formal Methods Europe (LNCS 2021), Berlin, Germany (2001)
10. Endres, T.: Java Tar 2.5. `http://www.trustice.com` (2003)
11. Necula, G.C., Rahul, S.P.: Oracle-based checking of untrusted software. In: Principles of Programming Languages, London, England (2001)
12. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall (1993)
13. Gupta, R.: Optimizing array bound checks using flow analysis. ACM Letters on Programming Languages and Systems **2** (1993) 135–150
14. Crary, K., Weirich, S.: Resource bound certification. In: Principles of Programming Languages, Boston, Massachusetts (2000)
15. Gong, L.: Inside Java 2 Platform Security. Addison-Wesley (1999)
16. Czajkowski, G., von Eicken, T.: JRes: a resource accounting interface for Java. In: Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, BC (1998)
17. Evans, D., Twyman, A.: Flexible policy-directed code safety. In: Security and Privacy, Oakland, California (1999)
18. Erlingsson, U., Schneider, F.: SASI enforcement of security policies: a retrospective. In: New Security Paradigms Workshop, Caledon, Canada (1999)
19. Colcombet, T., Fradet, P.: Enforcing trace properties by program transformation. In: Principles of Programming Languages, Boston, Massachusetts (2000)
20. Pandey, R., Hashii, B.: Providing fine-grained access control for Java programs via binary editing. Concurrency: Practice and Experience **12** (2000) 1405–1430
21. Chander, A., Mitchell, J., Shin, I.: Mobile code security by Java bytecode instrumentation. In: DARPA Information Survivability Confernce and Exposition. (2001)
22. Kim, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a run-time assurance tool for Java programs. Electronic Notes in Theoretical Computer Science **55** (2001)
23. Necula, G., Lee, P.: Safe kernel extensions without run-time checking. In: Operating Systems Design and Implementation, Seattle, Washington (1996)
24. Wallach, D., Appel, A., Felten, E.: SAFKASI: a security mechanism for language-based systems. Transactions on Software Engineering **9** (2000) 341–378
25. Patel, P., Lepreau, J.: Hybrid resource control of active extensions. In: Open Architectures and Network Programming, San Francisco, California (2003)
26. Vanderwaart, J., Crary, K.: Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, Carnegie-Mellon University (2004)