Safety Checking of Machine Code

Zhichen Xu Barton P. Miller Thomas Reps

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
{zhichen,bart,reps}@cs.wisc.edu

Abstract

We show how to determine statically whether it is safe for untrusted machine code to be loaded into a trusted host system.

Our safety-checking technique operates directly on the untrusted machine-code program, requiring only that the initial inputs to the untrusted program be annotated with typestate information and linear constraints. This approach opens up the possibility of being able to certify code produced by any compiler from any source language, which gives the code producers more freedom in choosing the language in which they write their programs. It eliminates the dependence of safety on the correctness of the compiler because the final product of the compiler is checked. It leads to the decoupling of the safety policy from the language in which the untrusted code is written, and consequently, makes it possible for safety checking to be performed with respect to an extensible set of safety properties that are specified on the host side.

We have implemented a prototype safety checker for SPARC machine-language programs, and applied the safety checker to several examples. The safety checker was able to either prove that an example met the necessary safety conditions, or identify the places where the safety conditions were violated. The checking times ranged from less than a second to 14 seconds on an UltraSPARC machine.

1 Introduction

Two prevailing trends in software development call for techniques to protect one software component from another. The first trend is *dynamic extensibility*, where a trusted host is extended by importing and executing *untrusted foreign code*. For example, web browsers download plug-ins [33,48]; databases load type-specific extensions [13,47]; operating systems load customized policies [2,11,23,34,38,41,50] and performance-measurement code [50]. There are even proposals for loading application-specific policies into Internet routers [52]. Certification of the safety of untrusted code is crucial in these domains. The second trend is *component-based software development*, where software components produced by multiple vendors are used to construct a complete application [6] (e.g., COM [21]). The component-based software-development model improves both software reusability and productivity. Because the software components can come from different sources, proper protection among software components is essential.

This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, NSF grants CDA-9623632, EIA-9870684, CCR-9625667, CCR-9619219, U.S.-Israel Binational Science Foundation grant 96-0037, a Vilas Associate Award form the University of Wisconsin, and DARPA contract N66001-97-C-8532. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

In this paper, we show how to determine statically whether it is safe for untrusted machine code to be loaded into a trusted *host* system. In contrast to work that enforces safety by restricting the things that can be expressed in a source language (e.g., safe languages, certifying compilers [31], and typed-assembly languages [25,26,28]), we believe that safe code can be written in any source language and produced by any compiler, as long as nothing "unsafe" is expressed in the machine code. This philosophical difference has several implications. First, it gives the code producer more freedom in choosing an implementation language. Instead of building a certifying compiler for each language, we can certify code produced by an off-the-shelf compiler. Second, it leads to the decoupling of a safety policy from the language in which the untrusted code is written. This makes it possible for safety checking to be performed with respect to an extensible set of safety properties that are specified on the host side.

In short, the most important, high-level characteristics of our safety-checking technique are (i) It operates directly on binary code; (ii) It provides the ability to extend the host at a very fine-grained level, in that we allow the foreign code to manipulate the internal data structures of the host directly; (iii) It enforces a default collection of safety conditions to prevent array out-of-bounds violations, address-alignment violations, uses of uninitialized variables, null-pointer dereferences, and stack-manipulation violations, in addition to providing the ability for the safety criterion to be extended according to an *access policy* specified by the host.

The essence of our approach is to recover source-level type information (more precisely, typestate information) based on a small amount of annotation about the initial inputs to the untrusted code, and then to apply some techniques that were originally developed for program verification to determine whether the untrusted code is safe. The initial annotation is in the form of typestates and linear constraints (i.e., linear equalities and inequalities that are combined with $\wedge$, $\vee$, $\neg$, and the quantifiers $\exists$ and $\forall$). Our analysis uses typestates (as opposed to types) because the condition under which it is safe to perform an operation is a function of not just the types of the operation's operands, but also their states. For example, it is safe to write to a location that stores an uninitialized value, but it is unsafe to read from it. Typestates differ from types by providing information at a finer granularity. Moreover, typestate checking [44,45] differs from traditional type checking in that traditional type checking is a flow-*insensitive* analysis, whereas typestate checking is a flow-*sensitive* analysis. Typestates can be related to *security automata* [3]. In a security automaton, all states are accepting states; the automaton detects a security-policy violation whenever read a symbol for which the automaton's current state has no transition defined. It is possible to design a typestate system that captures the possible states of a security automaton (together with a "security-violation" state). Typestate checking provides a method, therefore, for statically assessing whether a security violation might be possible.

To perform safety checking of machine-language programs, the issues we face include: (i) the design of a language for specifying policies; (ii) the inference of a typestate at each program point; (iii) overload resolution of certain machine-language instructions;[1] and (iv) synthesis of loop invariants. We use source-level typestates to describe the state of the host when untrusted code is to be invoked. A safety policy specifies the data that can be accessed and the functions (methods) that can be called by a piece of untrusted code. We extend typestate checking to infer a typestate for each program point, and to perform overload resolution. We use the induction-iteration method [49] to synthesize loop invariants.

The main contributions of this paper are as follows:

- Our technique opens up the possibility of being able to certify object code produced by off-the-shelf compilers (independent of both the source language and the compiler). We require only that the inputs to the untrusted code be annotated with typestate information and linear constraints.
- The technique is extensible: (i) in addition to a default collection of safety conditions that are always checked, additional safety conditions to be checked can be specified by the host; (ii) typestates can be related to security automata; this makes extending our technique to perform security checking natural.
- We extend the notion of typestate in several ways: (i) we use typestates to describe the state information of abstract locations in an abstract storage model; (ii) we extend typestates to include access permissions (which are used to specify the extent to which untrusted code is allowed to access host resources); (iii) in addition to using typestates to distinguish initialized values from uninitialized ones, we also use typestates to track pointers.
- Finally, by focusing only on enforcing fine-grained memory protection, we are able to use a decidable logic for expressing safety conditions and simple heuristics for synthesizing loop invariants. We wish to stress that, although we use techniques originally developed for verification of correctness, we are not trying to prove either total or partial correctness [9,12]. Safety checking is less ambitious than verification of correctness.

We have implemented a prototype safety checker for SPARC machine-language programs. We applied the safety checker to several examples (ranging from code that contains just a few branches, to code that contains nested loops, and to code with function and method calls). The safety checker was able to either prove an example met the necessary safety conditions, or identify the places where the safety conditions were violated, in times ranging from less than a second to 14 seconds on an UltraSPARC machine (see Section 6). Contrary to our initial intuition, we observed that certain compiler optimizations, such as loop-invariant code motion and improved register-allocation algorithms, actually make the task of safety checking easier.

The remainder of the paper is organized as follows: Section 2 describes the default safety properties that we enforce and the notion of a host-specified safety policy. Section 3 gives an overview of the safety-checking analysis. Section 4 describes the basis of the safety-checking analysis. Section 5 describes certain phases of the safety-checking analysis in greater detail. Section 6 presents our initial experience. Section 7 discusses related work. Section 8 discusses the limitations of our technique.

2 Safety Properties and Policies

When untrusted code is to be imported into a host system, we need to specify acceptable behaviors for the untrusted code. These behavior specifications take the form of safety conditions that include a collection of *default safety conditions* and *host-specified access policies*. The default safety conditions check for array out-of-bounds violations, address-alignment violations, uses of uninitialized values, null-pointer dereferences, and stack-manipulation violations.

The inputs to our safety-checking analysis include the untrusted code, a *host-typestate specification*, an *invocation specification*, and a *safety policy.* All inputs except for the untrusted code are provided by the host.

- A host-typestate specification includes (i) a data aspect that describes the type and the state of host data before the invocation of the untrusted code, and (ii) a control aspect that provides safety pre- and post-conditions for calling host functions and methods (in terms of the types and states of the parameters and return values, and linear constraints on them).
- An invocation specification provides information about the initial values passed to the untrusted code when it is invoked by the host.
- The host-specified access policy specifies the host data that can be accessed by a piece of untrusted code, and the host functions (methods) that can be called.

In our model, we view any addresses passed to a piece of untrusted code as doors into the host data region. A safety policy controls the memory locations (resources) that are accessible by specifying the pointer types that can be followed. For the memory locations reachable, the safety policy specifies the ways they can be accessed in terms of the types of the memory locations and their contents.

A policy is specified by (i) a classification of the memory locations into regions, and (ii) a list of triples of the form [Region : Category : Access Permitted]. A Region can be as large as an entire address space or as small as a single variable. The Category field is a set of types or aggregate fields. The Access field can be any subset of *r*, *w*, *f*, *x*, and *o*, meaning readable, writable, followable, executable, and operable, respectively. In our model, *r* and *w* are properties of a location, whereas *f*, *x*, and *o* are properties of the value stored in a location. The access permission *f* is introduced for pointer-typed values to indicate whether the pointer can be dereferenced. The access permission *x* applies to values of type "pointer to function" (i.e., values that hold the address of a function) to indicate whether the function pointed to can be called by the untrusted code. The access permission *o* includes the rights to

1. Instructions such as `add` and `ld` are overloaded; for example, `add` can be adding two integers or adding the base address of an array and an array index.

Untrusted Code

```
1:  mov %o0,%o2              ! move %o0 into %o2
2:  clr %o0                  ! set %o0 to zero
3:  cmp %o0,%o1              ! compare %o0 and %o1
4:  bge 12                   ! branch to 12 if %o0 ≥ %o1
5:  clr %g3                  ! set %g3 to zero
6:  sll %g3, 2,%g2           ! %g2 = 4 x %g3
7:  ld  [%o2+%g2],%g2        ! load from address %o2+%g2
8:  inc %g3                  ! %g3  = %g3 + 1
9:  cmp %g3,%o1              ! compare %g3 and %o1
10: bl 6                     ! branch to 6 if %g3 < %o1
11: add %o0,%g2,%o0          ! %o0 = %o0 + %g2
12: retl
13: nop
```

Host Typestate	Safety Policy	Invocation

`e`: <int, initialized, ro>
`arr`: <int [n], {`e`}, rfo>
{`n`≥`1`}
	V = {`e`, `arr`}
	[V : int : ro]
	[V : int [n] : rfo]
		`%o0` ← `arr`
		`%o1` ← `n`

`arr` is an integer array of size `n`, where `n`≥`1`. `e` is an abstract location that summarizes all elements of `arr`.	*`arr` and `e` are in the V region. All integers in the V region are readable and operable. All base addresses to an integer array of size `n` in the V region are readable, operable, and followable.*	*`arr` and the size of `arr` will be passed through the registers `%o0`, and `%o1`, respectively.*

Figure 1: A Simple Example: Summing the Elements of an Integer Array

"examine", to "copy", and to perform other operations not covered by *x* and *f*.

To get a feel for what a safety policy looks like, suppose that a user is asked to write an extension (as a piece of untrusted code) that finds out the lightweight process on which a thread is running, and suppose that information about threads is stored in the host address space in a linked list defined by the structure `thread`

```
        struct thread {
            int tid;
            int lwpid;
            ...
            struct thread * next;
        };
```

The following policy allows the extension to read and examine the `tid` and `lwpid` fields, and to follow only the `next` field (*H* stands for "Host Region", which is the region in which the list of threads is stored):

[*H* : `thread.tid`, `thread.lwpid` : *ro*]
[*H* : `thread.next` : *rfo*]

The above model can be used to specify a variety of different safety policies. For example, we can specify something roughly equivalent to *sandboxing* [55]. The original sandboxing model partitions the address space into protection domains, and modifies a piece of untrusted code so that it accesses only its own domain. In our model, sandboxing boils down to allowing untrusted code to access memory only via valid addresses in the untrusted data region, but otherwise to examine, and operate on data items of any type. Because an address of a location in the host region cannot be dereferenced, side-effects are confined to the untrusted region. Our approach differs from sandboxing in that it is purely static, and it does not make any changes to the untrusted code.

While sandboxing works well in situations where it is appropriate to limit memory accesses to only the untrusted data region, forbidding access to all data in the host region is often too draconian a measure. For instance, access to the host data region is necessary for applications as simple as performance instrumentation (e.g., to read statistics maintained by the host environment). In our model, more aggressive policies are defined by allowing simple reads and writes to locations in the host data region, but forbidding pointers to be followed or modified. We can go even further by specifying policies that permit untrusted code to follow certain types of valid pointers in the host data region in order to traverse linked data structures. We can even specify more aggressive policies that permit untrusted code to change the shape of a host data structure, by allowing the untrusted code to modify pointers.

What we have presented here is a simplified view of a safety policy. In reality, a safety policy can also include a safety postcondition (typestates and linear constraints) for ensuring that certain invariants defined on the host data are restored by the time control is returned to the host.

3 Overview of Safety-Checking Analysis

Our goal is to verify that untrusted code (i) obeys the collection of default safety conditions, and (ii) accesses data and calls host functions only in a manner that is consistent with a given safety policy. To do this, we introduce an abstract storage model, use the default safety conditions and host-specified access policy to attach a safety predicate to each instruction, use static analysis to determine an approximation to the contents of memory locations at each point in the program, and check whether each instruction obeys the corresponding safety predicate. The abstract storage model includes the notion of abstract locations and typestates. A typestate describes the type, state, and access permissions of the value stored in an abstract location.

We divide the safety-checking analysis into five phases: preparation, typestate propagation, annotation, local verification, and global verification. We now illustrate these phases informally by means of a simple example. Figure 1 shows a piece of untrusted code (in SPARC assembly language) that sums the elements of an integer array. It also shows the host-typestate specification, the safety policy, and the invocation specification.

The host-typestate specification states that `arr` is an integer array of size `n`, where `n` ≥ `1`. We have used a single abstract location `e` to summarize all elements of the array `arr`. The safety policy states that `arr` and `e` are in the *V* region, that all integers in the *V* region are readable and operable, and that all base addresses to an integer array of size `n` in the V region are readable, operable, and followable. The invocation specification states that `arr` and the size of `arr` will be passed through the registers `%o0` and `%o1`, respectively. The code uses three additional registers, `%o2`, `%g2`, and `%g3`.

Phase 1: *Preparation* takes the host-typestate specification, the safety policy, and the invocation specification, and translates them into *initial annotations* that consist of linear constraints and the typestates of the inputs.

For the example given in Figure 1, the initial annotations are shown in Figure 2. The fact that the address of `arr` is passed via register `%o0` is described in the second line in column 1. The fact that the size of `arr` is passed via the register `%o1` is captured by

Initial Annotations	
Initial Typestate	**Initial Constraints**
`e`:<`int`, initialized, ro> `%o0`:<`int` [n], {`e`}, rwfo> `%o1`:<`int`, initialized, rwo>	`n ≥ 1 ∧ n=%o1`

Figure 2: Initial Annotations

the linear constraint "`n=%o1`". Note that `%o0` and `%o1` both have the *r* and *w* access permissions. These refer to the registers themselves (i.e., the untrusted code is permitted to read and change the value of both registers). However, array `arr` cannot be overwritten because the access permission for `e`, which acts as a surrogate for all elements of `arr`, does not have the *w* permission.

Phase 2: *Typestate propagation* takes the untrusted code and the initial annotations as inputs. It annotates each instruction with an abstract representation of the memory contents that characterize the state before the execution of that instruction. For our example, this phase discovers that the instruction at line 7 is an array access, where `%o2` holds the base address of the array and `%g2` represents the index.

Phase 3: *Annotation* takes as input the typestate information discovered in Phase 2, and traverses the untrusted code to annotate each instruction with local and global safety preconditions and with assertions. The local safety preconditions are conditions that can be checked using typestate information alone. The assertions are facts that can be derived from the results of typestate propagation. For our example, the assertions, local safety preconditions, and global safety preconditions for the instruction at line 7 are summarized in Figure 3.

Phase 4: *Local verification* checks the local safety preconditions. (In our example, the local safety preconditions are all true at line 7.)

Assertions	Local Safety Preconditions	Global Safety Preconditions
%o2 is the address of an integer array `%o2` mod 4 = 0 `%o2` ≠ `NULL`	`e` is readable; `%g2` is writable; `%o2` is followable, and operable	*Array bounds checks* `%g2`≥ 0 ∧ `%g2` < 4n ∧`%g2` mod 4 = 0 *Alignment and null-pointer checks* `%o2` ≠ `NULL` ∧ (`%o2` + `%g2`) mod 4 = 0

Figure 3: Assertions and Safety Preconditions for Line 7

Phase 5: *Global verification* attempts to verify the global safety preconditions using program-verification techniques. In the presence of loops, we use the induction-iteration method to synthesize loop invariants. In proving that at line 7 index `%g2` is less than the upper bound `n`, our safety checker automatically synthesizes the loop invariant "`n` > `%g3` ∧ `n` ≥ `%o1`".

4 The Basis of Safety-Checking Analysis

This section describes (i) the abstract storage model used in our safety-checking analysis, (ii) an abstract operational semantics for SPARC machine-language instructions, and (iii) how safety predicates are attached to instructions.

4.1 An Abstract Storage Model

The abstract storage model we use includes the notion of abstract locations and typestates. An *abstract location* summarizes a set of physical locations so that the safety-checking analysis has a finite-size domain to work over. (In general, the number of concrete activation records is unbounded in the presence of recursion, as are the number of concrete objects allocated in a loop and the size of concrete linked data-structures.) An abstract location has a name, a size, an alignment, and optional attributes *r* and *w* to indicate whether the location is readable and writable, respectively.

A *typestate* records properties of abstract locations. A typestate is defined by a triple <type, state, access>. We define a meet operation ⊓ on typestates so that typestates form a meet semi-lattice. The meet of two typestates is defined as the meet of their respective components.

t :: ground		*Ground types*
|	abstract	*Abstract types*
|	$t\,[n]$	*Pointer to the base of an array of type t of size n*
|	$t\,(n)$	*Pointer into the middle of an array of type t of size n*
|	*t* `ptr`	*Pointer to t*
|	$s\,\{m_1, ..., m_k\}$	*`struct`*
|	$u\,\{\|m_1, ..., m_k\|\}$	*`union`*
|	$(t_1, ..., t_k) \rightarrow t$	*Function*
m:: (*t*, *l*, *i*)		*Member labeled l of type t at offset i*
ground:: `int8` | `uint8` | `int16` | `uint16` | `int32` | ...		

Figure 4: A Simple Type System

t *stands for type, and* m *stands for a struct or union member.*

Our type system is based on that of Siff *et al* [42], with the addition of abstract types, pointers into the middle of arrays, and alignment and size constraints on types. (See Figure 4; alignment and size constraints on types are not shown.) The type $t(n)$ denotes a pointer that points somewhere into the middle of an array of type *t* of size *n*. The meet operation on types are defined as follows. The meet of two different non-pointer types is $\perp_t$.[2] The meet of two different pointer types, or the meet of a pointer type and a non-pointer type is $\perp_t$. The meet of type $t[n]$ and type $t(n)$ is $t(n)$, whereas the meet of $t[n]$ and $t[m]$ or the meet of $t(n)$ and $t[m]$ is $\perp_t$ if $m \neq n$.

The state component of a typestate captures the notion of an object of a given type being in an appropriate state for some operations but not for others. The state lattice contains a bottom element $\perp_s$ that denotes an undefined value of any type. Figure 5 illustrates selected elements of the state lattice. Since we also use the state descriptors to track abstract locations that represent pieces of stack- and heap-allocated storage, they resemble the storage-shape graphs of Chase *et al* [5].

An access permission is either a subset of {*f*, *x*, *o*}, or a tuple of access permissions. If an abstract location stores an aggregate, its access permission will be a tuple of access permissions, with the elements of the tuple denoting the access permissions of the

2. Our implementation also incorporates a notion of subtyping for ground types. This makes the analysis more precise when dealing with operations such as "load byte" and "load half word".

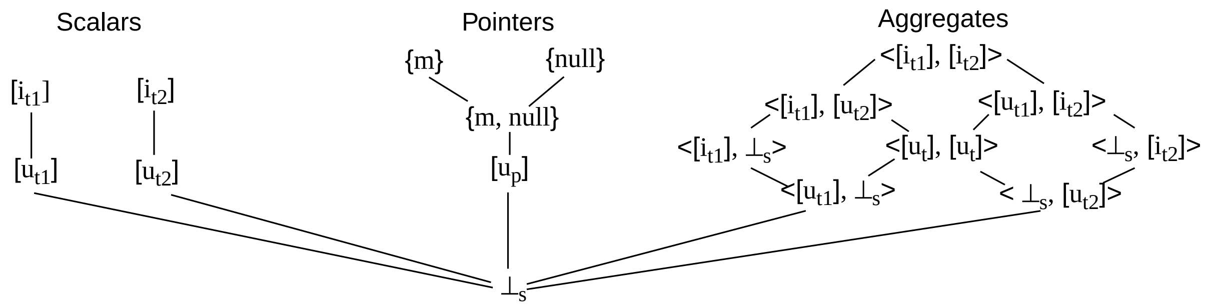


Figure 5: A Portion of the State Lattice

For a scalar type t, its state can be $[u_t]$ or $[i_t]$, which denote uninitialized and initialized values, respectively. For a pointer type p, its state can be $[u_p]$, which is the state of an uninitialized pointer, or P, a non-empty set of abstract locations referenced, where one of the elements of P can be null. *For sets P_1 and P_2, we define $P_1 \leq P_2$ iff $P_2 \subseteq P_1$. For an aggregate type G, its state is given by the states of its fields.*

respective aggregate fields. The meet of two access-permission sets is their intersection. The meet of two tuples of access permissions is given by the meet of their respective elements.

The reader may be puzzled why a safety policy is defined in terms of five kinds of access permissions (*r, w, f, x,* and *o*), whereas typestates have only three kinds (*f, x,* and *o*). The reason is that *f, x,* and *o* are properties of a *value*, whereas *r* and *w* are properties of a *location*. Typestates capture properties of *values*. Policies specify the *r* and *w* permissions of abstract locations, as well as *f, x,* and *o* permissions of their values.

The typestate lattice also includes a top element T.

In the remainder of the paper, we use *absLoc* to denote the set of all abstract locations, and the symbols *l* and *m* to denote individual abstract locations. We use *Size*(*l*), *Align*(*l*), to denote the size, and alignment, respectively, of abstract location *l*. We call an abstraction location that summarizes more than one physical location a *summary location*. A register is always readable and writable, and has an alignment of zero. A constant always has access permission *o*.

4.2 An Abstract Operational Semantics for SPARC Machine Instructions

An abstract store is given by a total map *M*: *absLoc* → typestate. We define the abstract operational semantics of a SPARC machine instruction as a transition function *R: M*→*M*. We use T(*l*), S(*l*), and A(*l*) to denote the type, state, and access component of the typestate of abstract location *l*, respectively.

4.2.1 Overload Resolution

We determine an appropriate typestate for each abstract location at each program point by finding the greatest fixed point of a set of typestate-propagation equations (see Section 4.2.2). Overload resolution of instructions such as `add` and `ld` falls out as a by-product of this process: The type components of the typestates obtained for the arguments of overloaded instructions let us identify whether a register holds a scalar, a pointer, or the base address of an array (and hence whether an instruction such as `add %o0,%g2,%o0` represents the addition of two scalars, a pointer indirection, or an array-index calculation). To achieve this, we define the abstract operational semantics of SPARC machine instructions to be strict in T. Consequently, during typestate checking, propagation of information through the instructions of a loop is delayed until a non-T value arrives at the loop entrance.

One artifact of this method is that each occurrence of an overloaded instruction is resolved to just a single usage kind (e.g., scalar addition, pointer indirection, or array-index calculation). We call this the *single-usage restriction*. We believe that this restriction does not represent a significant limitation in practice because we are performing typestate checking (which is flow sensitive). For example, typestate checking allows an instruction such as `add %o0,%g2,%o0` to be resolved as a pointer indirection at one occurrence of the instruction, but as an array-index calculation at a different occurrence.

Operation	Assumption	Type-Propagation Rule	State-Propagation Rule	Access-Propagation Rule
1 add $\mathbf{r}_s$, *Opnd*, $\mathbf{r}_d$	Scalar add	1. $T'(\mathbf{r}_d) = T(\mathbf{r}_s) \sqcap T(Opnd)$. 2. for $l \neq \mathbf{r}_d$, $T'(l) = T(l)$.	1. $S'(\mathbf{r}_d) = S(\mathbf{r}_s) \sqcap S(Opnd)$. 2. for $l \neq \mathbf{r}_d$, $S'(l) = S(l)$.	1. $A'(\mathbf{r}_d) = A(\mathbf{r}_s) \cap A(Opnd)$. 2. for $l \neq \mathbf{r}_d$, $A'(l) = A(l)$.
2	Array-index calculation $T(\mathbf{r}_s) = t\,[n]$	1. $T'(\mathbf{r}_d) = t\,(n]$. 2. for $l \neq \mathbf{r}_d$, $T'(l) = T(l)$.	1. $S'(\mathbf{r}_d) = S(\mathbf{r}_s)$. 2. for $l \neq \mathbf{r}_d$, $S'(l) = S(l)$.	1. $A'(\mathbf{r}_d) = A(\mathbf{r}_s)$. 2. for $l \neq \mathbf{r}_d$, $A'(l) = A(l)$.
3 st $\mathbf{r}_s$, $[\mathbf{r}_a+n]$	Store to an aggregate field Let $F = \{s.\beta \mid s \in S(\mathbf{r}_a),$ $\beta \in lookUp(T(s), n, 4)\}$	1. if $F=\{l\}$, if l is not a summary location, $T'(l)=T(\mathbf{r}_s)$; otherwise $T'(l)=T(\mathbf{r}_s) \sqcap T(l)$. 2. if $\mid F \mid > 1$, for $l \in F$, $T'(l)=T(\mathbf{r}_s) \sqcap T(l)$. 3. for $l \notin F$, $T'(l) = T(l)$.	1. if $F=\{l\}$, if l is not a summary location, $S'(l)=S(\mathbf{r}_s)$; otherwise $S'(l)=S(\mathbf{r}_s) \sqcap S(l)$. 2. if $\mid F \mid >1$, for $l \in F$, $S'(l)=S(\mathbf{r}_s) \sqcap S(l)$. 3. for $l \notin F$, $S'(l) = S(l)$.	1. if $F = \{l\}$, if l is not a summary location, $A'(l)=A(\mathbf{r}_s)$; otherwise $A'(l) =A(l) \cap A(\mathbf{r}_s)$. 2. if $\mid F \mid >1$, for $l \in F$, $A'(l) =A(l) \cap A(\mathbf{r}_s)$. 3. for $l \notin F$, $A'(l) = A(l)$.

Table 1: Propagation of Type, State, and Access information

Operation	Assumptions	Local Safety Predicates	Global Safety Predicates
1 add $\mathbf{r}_s$, *Opnd*, $\mathbf{r}_d$	Scalar add	$operable(\mathbf{r}_s) \wedge operable(Opnd)$	
2 add $\mathbf{r}_s$, *Opnd*, $\mathbf{r}_d$	Array-index calculation $T(\mathbf{r}_s) = t\,[n]$	$operable(\mathbf{r}_s) \wedge operable(Opnd)$	$\text{null} \notin S(\mathbf{r}_s) \wedge inbounds(sizeof(t), 0, n, Opnd)$
4 st $\mathbf{r}_s$, $[\mathbf{r}_a+n]$	Store to an aggregate field Let $F = \{s.\beta \mid s \in S(\mathbf{r}_a), \beta \in lookUp(T(s), n, 4)\}$	$followable(\mathbf{r}_a) \wedge operable(\mathbf{r}_a) \wedge F \neq \emptyset \wedge \text{forall } l \in F, assignable(\mathbf{r}_s, l)$	$\text{null} \notin S(\mathbf{r}_a) \wedge$ forall $a \in S(\mathbf{r}_a)$, $align(Align(a)+n, 4) \wedge sizeof(T(\mathbf{r}_s))=4$

Table 2: Attachment of Safety Properties

4.2.2 Propagation of Type, State, and Access Information

For the sake of brevity, Table 1 shows the rules for propagating type, state, and access information only for two different kinds of uses of the `add` instruction (scalar add and array-index calculation) and for storing to an aggregate field. $\mathbf{r}_s$, $\mathbf{r}_a$, and $\mathbf{r}_d$ are registers, and *Opnd* is either an integer constant *n* or a register. We use $l \neq r_d$ to denote $l \in (absLoc - \{r_d\})$, and use $T(l)$ and $T'(l)$ to denote the types of abstract location *l* before and after the execution of an instruction, respectively. We define *S(l)*, *S'*(*l*), *A(l)*, and *A'*(*l*) similarly. We use β to refer to a (possibly empty) sequence of field names. The function *lookUp* takes a type and two integers *n* and *m* as input; it returns the set of fields that are at offset *n* and of size *m*, or ∅ if no such fields exist.

1. The typestate-propagation rules for scalar-add state that after the execution of the `add` instruction, the typestate of $\mathbf{r}_d$ is the meet of those of $\mathbf{r}_s$ and *Opnd* before the execution, and the typestate of all other abstract locations in *absLoc* remain unchanged.
2. For an array-index calculation, the type of the destination register becomes "*t*(*n*]", where "*t*" is the type of an array element. The type "*t*(*n*]" indicates that $\mathbf{r}_d$ could point to any element in the array. As to the state-propagation rule, at present we use a single abstract location to summarize the entire array; thus the state of the destination register is the same as that of the source register.
3. The typestate-propagation rules for storing to an aggregate field are divided into two cases, depending on whether strong or weak update is appropriate. The abstract-location set *F* gives the set of locations into which the `st` instruction may store. The pointer "$\mathbf{r}_a+n$" must point to *l* (∈ *F*), if $|F|$=1 and *l* is not a summary location. In this case, *l* receives the typestate of the source register. The pointer "$\mathbf{r}_a+n$" may point to the locations in *F*, if $|F|>1$, or $F=\{l\}$ and *l* is a summary location. In this case, each possible destination receives the meet of the typestate before the operation and the typestate of the source register.

4.3 Attachment of Safety Predicates

Phase 3 of the safety-checking analysis uses the default safety conditions, the host-specified access policy, and the results of overload resolution to attach a collection of safety predicates to each instruction. These safety predicates are divided into local safety predicates and global safety predicates, depending on whether or not the predicates can be validated using typestate information alone. Table 2 summarizes the safety predicates for the two cases of `add` and the one case of `st` that were described earlier.

1. For scalar `add`, the safety predicate specifies that uninitialized values must not be used. The predicate *readable*(*l*) is true *iff l* is readable, and the predicate *operable* is true *iff* $o \in A(l)$ and $S(l) \notin \{[u_{T(l)}], \perp_s\}$.
2. The safety predicates for an array-index calculation state that $\mathbf{r}_s$ and *Opnd* must both be readable and operable, and the index must be within the bounds of the array. The predicate *inbounds*(*size, low, high, i*) is true *iff* $low \times size \leq i < high \times size$, and $i \bmod size = 0$.
3. The safety predicates for `st` state that (i) $\mathbf{r}_a$ must be followable and *n* must be a valid index of a field of the right size; (ii) $\mathbf{r}_a$ must be non-null, and the address "$\mathbf{r}_a+n$" must be properly aligned. The predicate *followable*(*l*) is true *iff* $f \in A(l)$, and T(*l*) is a pointer type; the predicate *assignable*(*m, l*) is true *iff readable*(*m*), *writable*(*l*), and $(T(l) \leq T(m))$, $Align(l) \bmod Align(T(m))=0$ and $sizeof(T(m)) \leq Size(l))$ all hold; the predicate *align*(*A, n*) is true *iff* $\exists\, \alpha$ st. $A = n\alpha$.

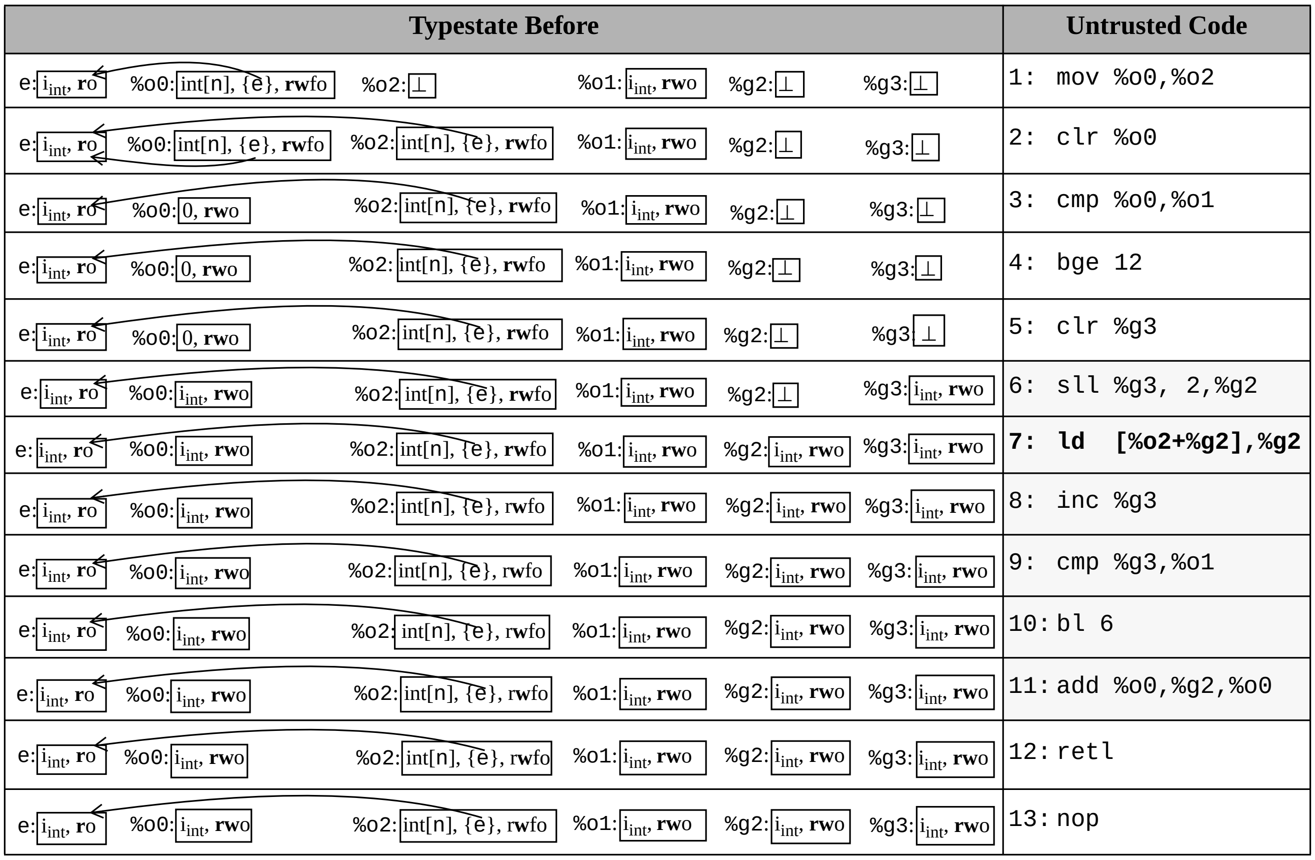
5 Elaboration of Phases 2 and 5 of Safety-Checking Analysis

In this section, we expand upon Phases 2 and 5 of the safety-checking analysis of the array-summation example introduced in Section 3.

5.1 Phase 2 — Typestate Propagation

The typestate-propagation phase works on interprocedural control-flow graphs, where the nodes in the graph represent instructions and the edges represent control flow in the usual way. To create a safe approximation of the program state before and after each node, each node has two total maps (each of type *absLoc* → typestate) representing abstract stores. The algorithm for typestate propagation is a standard worklist-based algorithm. It starts with the map $\lambda l.\top$ at all program points except for the entry node. The map for the entry node incorporates the initial annotations generated during Phase 1 (see Figure 2); the abstract locations for which there are no initial annotations are initialized with the typestate $<\perp_t, \perp_s, \emptyset>$.

Initially, the first instruction of the untrusted code is placed on the worklist. An instruction is chosen from the worklist and examined. The typestates of the abstract locations at the entry to the examined instruction become the meet of the corresponding typestates at the exits of the instruction's predecessors. The instruction is interpreted abstractly using the new typestates. This may cause the abstract store associated with the exit of the instruction to change. In that case, each instruction that is a successor of



Typestate Before	Untrusted Code
e: i_{int}, **r**o %o0: int[n], {e}, **rw**fo %o2: ⊥ %o1: i_{int}, **rw**o %g2: ⊥ %g3: ⊥	`1: mov %o0,%o2`
e: i_{int}, **r**o %o0: int[n], {e}, **rw**fo %o2: int[n], {e}, **rw**fo %o1: i_{int}, **rw**o %g2: ⊥ %g3: ⊥	`2: clr %o0`
e: i_{int}, **r**o %o0: 0, **rw**o %o2: int[n], {e}, **rw**fo %o1: i_{int}, **rw**o %g2: ⊥ %g3: ⊥	`3: cmp %o0,%o1`
e: i_{int}, **r**o %o0: 0, **rw**o %o2: int[n], {e}, **rw**fo %o1: i_{int}, **rw**o %g2: ⊥ %g3: ⊥	`4: bge 12`
e: i_{int}, **r**o %o0: 0, **rw**o %o2: int[n], {e}, **rw**fo %o1: i_{int}, **rw**o %g2: ⊥ %g3 ⊥	`5: clr %g3`
e: i_{int}, **r**o %o0: i_{int}, **rw**o %o2: int[n], {e}, **rw**fo %o1: i_{int}, **rw**o %g2: ⊥ %g3: i_{int}, **rw**o	`6: sll %g3, 2,%g2`
e: i_{int}, **r**o %o0: i_{int}, **rw**o %o2: int[n], {e}, **rw**fo %o1: i_{int}, **rw**o %g2: i_{int}, **rw**o %g3: i_{int}, **rw**o	**`7: ld [%o2+%g2],%g2`**
e: i_{int}, **r**o %o0: i_{int}, **rw**o %o2: int[n], {e}, r**w**fo %o1: i_{int}, **rw**o %g2: i_{int}, **rw**o %g3: i_{int}, **rw**o	`8: inc %g3`
e: i_{int}, **r**o %o0: i_{int}, **rw**o %o2: int[n], {e}, r**w**fo %o1: i_{int}, **rw**o %g2: i_{int}, **rw**o %g3: i_{int}, **rw**o	`9: cmp %g3,%o1`
e: i_{int}, **r**o %o0: i_{int}, **rw**o %o2: int[n], {e}, r**w**fo %o1: i_{int}, **rw**o %g2: i_{int}, **rw**o %g3: i_{int}, **rw**o	`10: bl 6`
e: i_{int}, **r**o %o0: i_{int}, **rw**o %o2: int[n], {e}, r**w**fo %o1: i_{int}, **rw**o %g2: i_{int}, **rw**o %g3: i_{int}, **rw**o	`11: add %o0,%g2,%o0`
e: i_{int}, **r**o %o0: i_{int}, **rw**o %o2: int[n], {e}, r**w**fo %o1: i_{int}, **rw**o %g2: i_{int}, **rw**o %g3: i_{int}, **rw**o	`12: retl`
e: i_{int}, **r**o %o0: i_{int}, **rw**o %o2: int[n], {e}, r**w**fo %o1: i_{int}, **rw**o %g2: i_{int}, **rw**o %g3: i_{int}, **rw**o	`13: nop`

Figure 6: Results of Typestate Propagation

the examined instruction is added to the worklist. This process is repeated until the worklist is empty.

Figure 6 shows the results of typestate propagation applied to our running example. The right column shows the instructions. The left column shows the abstract store before the execution of the corresponding instruction. Lines 6 to 11 correspond to the loop. Initially, `%o0` holds the base address of the integer array `arr` (whose elements are summarized by `e`), and `%o1` holds the size of the array. Typestate checking is initiated by placing the `mov` instruction at line 1 on the worklist. Abstract interpretation of the `mov` instruction at line 1 sets the contents of `%o2` to point to `e`. Because the typestate of `%o2` has changed, the instruction at line 2 is placed on the worklist. The interpretation of the `clr` instruction at line 2 sets the contents of `%o0` to 0. This process continues until the worklist becomes empty. For line 7, the results show that `%o2` holds the base address of an integer array and that `%g2` is an integer (and hence must be an index).

5.2 Phase 5 —Verification of Global Safety Preconditions

The fifth phase verifies the global safety preconditions using program-verification techniques. This phase involves (i) generating verification conditions (VCs), and (ii) verifying the VCs using a theorem prover. Unlike standard techniques for program verification, in which one monolithic VC is created containing all properties to prove, we check the validity of the global safety preconditions in a demand-driven fashion, and verify the conditions one at a time.

Since array-bounds, null-pointer, and address-alignment requirements can usually be represented as linear equalities and inequalities, our theorem prover is based on the Omega Library [17]. The Omega library represents relations and sets as Presburger formulas, which are formulas constructed by combining affine constraints on integer variables with the logical operations ¬, ∧, and ∨, and the quantifiers ∀ and ∃. The affine constraints can be either equality or inequality constraints [17]. Presburger formulas are decidable. For more details on the Omega Library, see Kelly *et al* [17] and Pugh *et al* [35,36,37].

When the untrusted code involves loops, an additional step is needed to synthesize loop invariants. In our system, the synthesis of loop invariants is attempted by means of the induction-iteration method [49]. We have extended the induction-iteration method to synthesize loop invariants for natural loops (and also to work on machine-language programs).

5.2.1 Induction Iteration Method

The induction-iteration method uses the “weakest liberal precondition” (wlp) as a heuristic for generating loop invariants. The weakest liberal precondition of statement S with respect to postcondition Q, denoted by wlp(S, Q), is a condition R such that if statement S is executed in a state satisfying R, (i) Q is always true after the termination of S (if S terminates), and (ii) no condition weaker than R satisfies (i). A weakest liberal precondition differs from a weakest precondition in that a weakest liberal precondition does not guarantee termination. Since our technique works on machine language programs, we have extended the induction-iteration method to work on *reducible* control-flow graphs [29], and partition the control-flow graph into code regions that are either cyclic (*natural loops*) or acyclic.

The method for generating wlps for non-conditional instructions is the same as those for generating weakest preconditions [9]. We compute the wlp of load and store instructions based on Morris's general axiom of assignment [24], which provides a general framework for computing weakest precondition for assignments to pointer-typed variables. To compute the wlp for an acyclic code region, the standard technique for verification generation is used.

To compute the wlp for a natural loop, we define W(0) as the wlp generated by back-substituting the postcondition Q to be proved until the entry of a loop is reached, i.e., W(0) = wlp(loop-body, Q), and define W($i+1$) as wlp(loop-body, W(i)). The wlp of the loop is the formula $\bigwedge_{i\geq 0}$W(i).

We use L(j) to denote $\bigwedge_{j\geq i\geq 0}$W($i$). The induction-iteration method attempts to find an L(j) that is both (i) true on entry to the loop and (ii) a loop invariant (i.e., L(j) implies wlp(loop-body, L(j))). Suzuki and Ishihata show that this can be established by showing:

L(j) is true on entry to the loop, and (**Inv.0**(j))

L(j) $\supset$ W($j+1$) (**Inv.1**(j)).

Their argument runs as follows [49]:

1. From the assumption that L(j) implies W(j+1), we know that $\bigwedge_{j\geq i\geq 0}$W($i$) implies $\bigwedge_{j\geq i\geq 0}$W($i$+1)**.**

2. Next, we observe that $\bigwedge_{j\geq i\geq 0}$W($i$+1) is equivalent to wlp(loop-body, L(j))

   $\bigwedge_{j\geq i\geq 0}$ W(i+1) = $\bigwedge_{j\geq i\geq 0}$wlp(loop-body, W($i$))

   = wlp(loop-body, $\bigwedge_{j\geq i\geq 0}$W($i$)) = wlp(loop-body, L(j))

```
Induction_Iteration() : SUCCESS | FAILURE {
i=0; Create formula W(0); //Try L(-1)
while (i < MAX_NUMBER_OF_ITERATIONS) {
  switch (Theorem_prover(($\bigwedge_{i-1 \geq k \geq 0}$W (k)) $\supset$ W (i))) { //inv.1(i-1)
    TRUE:         return SUCCESS;
    OTHERWISE:{ //Try L(i)
      switch (Theorem_prover(wlp(<on-entry-to-loop>,W(i)))) {//inv.0(i)
        TRUE:         W(i+1)=wlp(loop-body, W(i));
                      i=i+1;
        OTHERWISE:    return FAILURE;
      }
    }
  }
}
}
```

Figure 7: The Basic Induction-Iteration Algorithm

The induction iteration method, in essence, iterates the following steps: create the expression L(j) as the current candidate for the loop invariant; generate VCs for (**Inv.0**(j)) and (**Inv.1**(j)); attempt to verify the VCs using a theorem prover. Figure 7 shows the basic induction-iteration algorithm taken from Suzuki and Ishihata (rewritten in pseudo code) [49].

The reader may be puzzled why the algorithm first tests for **inv.1**(i-1), and then tests for **inv.0**(i). This is because the test **inv.0**(i-1) that matches the test for **inv.1**(i-1) is performed in the previous iteration. In the case of L(-1), **inv.0**(-1) holds vacuously because L(-1)=$\bigwedge_{-1\geq i\geq 0}$W($i$) = true.

We have made several enhancements to the basic induction-iteration algorithm. The first enhancement is the ability to deal with multiple loops. To ensure that the induction-iteration algorithm will terminate in the presence of nested loops, we extended the basic induction-iteration algorithm to treat an inner loop differently when trying to verify **Inv.0**. In the case of computing a wlp for an inner loop due to the synthesis of a loop invariant for an outer loop, instead of naively testing that W(i) of the inner loop is true on entry to the loop, we record the current trial invariant L(j) of the outer loop, and first try to verify that L(j) implies W(i).

Second, procedure calls complicate the induction-iteration method in three ways: (i) handling a procedure call when performing a back-substitution, (ii) reaching the procedure entry before we can prove or disprove a condition, and (iii) recursion. To handle procedure calls during back-substitution, we simply walk through the body of the callee as through it is inlined in the caller; this will generate a wlp for the callee function with respect to the postcondition that is propagated to the callsite. When we reach the entry of a procedure, we check that the conditions are true at each callsite by using these conditions as postconditions to be proven at each of the caller. To simplify matters, our present system detects and rejects recursive programs. In principle, we could use the induction-iteration method to synthesize invariant entry conditions for recursive functions as we do for loops.

Third, certain conditionals in a loop can sometimes weaken L(j) to such an extent that it is prevented from becoming a loop-invariant. To address this problem, we strengthen L(j) by computing the disjunctive normal form of wlp(loop-body, W($i-1$)), and try each of its disjuncts as W(i) in turn. We rank the candidates according to a simple heuristic and test the potential candidates for W(i) using a breadth-first strategy.

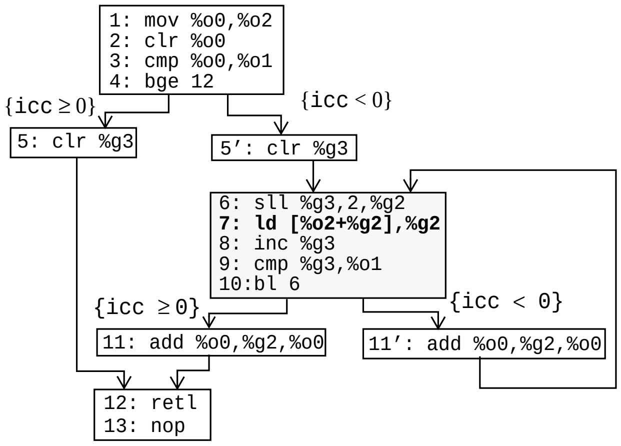
Fourth, the breath-first strategy of the extended induction-iteration algorithm also incorporates a technique called *generalization,* also introduced by Suzuki and Ishihata [49]. The generalization of a formula f is defined as "$\neg$(*elimination*($\neg f$))." Elimination uses the Fourier-Motzkin variable-elimination method to eliminate variables from the set of variables in $\neg f$ to generate a simplified set of constraints that has the same integer solution as f. If there are several resulting generalizations, then each of them in turn is chosen to be the generalized formula.

Fifth, the conditionals in the program can cause the formula under consideration to blow-up in size exponentially during VC generation. To reduce this effect, back-substitution over a region is performed in backwards topological order (with respect to the program's control-flow graph), and the formula at each junction point is simplified. This strategy effectively controls the size of the formulas considered, and ultimately the time that is spent in the theorem prover.

Finally, to reduce the number of times the induction-iteration algorithm is performed, we back-substitute all formulas to be proven until they reach a loop entry. We partition the formulas into groups that are made of comparable constituents, and invoke the induction-iteration algorithm only for the strongest formulas in each group.

5.2.2 Example

Here we illustrate how the induction-iteration method is applied in our running example. The control-flow graph of the program is shown in Figure 8. The instructions at lines 5 and 11 are replicated to model the semantics of delayed branches. We use a single boolean variable `icc` to model the SPARC condition code and label each control-flow graph edge with the condition for that edge to be taken. Below, we use the line number of an instruction to denote the instruction, and a sequence of line numbers within square brackets to represent a path.



```
1: mov %o0,%o2
2: clr %o0
3: cmp %o0,%o1
4: bge 12

5: clr %g3

5′: clr %g3

6: sll %g3,2,%g2
7: ld [%o2+%g2],%g2
8: inc %g3
9: cmp %g3,%o1
10:bl 6

11: add %o0,%g2,%o0

11′: add %o0,%g2,%o0

12: retl
13: nop
```

Figure 8: Induction Iteration: Example

To verify that “%g2 < 4n” holds at line 7, we perform back-substitution, starting with “%g2 < 4n”. Back-substituting this condition across line 6 produces “%g3 < n”. Since the instruction at line 6 is the entry of a natural loop, we attempt to synthesize a loop invariant that implies “%g3 < n”.

We set W(0) to “%g3 < n”. Since W(0) is not a tautology, we need to verify that W(0) is true on entry to the loop and to create the formula for W(1). The fact that W(0) is true on entry to the loop can be shown by back-substituting W(0) along the path [5’,4,3,2,1]. To create W(1), we perform back-substitution through the loop body, starting with the formula “%g3 < n”, until we reach the loop entry again:

1. wlp([11’], “%g3 < n”) = “%g3 < n”
2. wlp([11’,10], “%g3 < n”) = “icc < 0 ⊃ %g3 < n”
3. wlp([10,9], “icc < 0 ⊃ %g3 < n”)
   = “%g3 < %o1 ⊃ %g3 < n”
4. wlp([8], “%g3 < %o1 ⊃ %g3 < n”)
   = “%g3+1 < %o1 ⊃ %g3+1 < n”
5. wlp([7,6], “%g3+1 < %o1 ⊃ %g3+1 < n”)
   = “%g3+1 < %o1 ⊃ %g3+1 < n”

Thus, W(1) is the formula “%g3+1 < %o1 ⊃ %g3+1 < n”.

Unfortunately, W(0) ⊃ W(1) is not a tautology. Instead of continuing by creating W(2) etc., we strengthen W(1) using the generalization technique mentioned in Section 5.2.1. The steps to generalize W(1) are as follows: (i) negating “%g3+1 < %o1 ⊃ %g3+1 < n” produces “%g3+1 < %o1 ∧ %g3+1 ≥ n”; (ii) eliminating %g3 produces “%o1 > n”; (iii) negating “%o1 > n” produces “%o1 ≤ n”. Consequently, we set W(1) to be the generalized formula “%o1 ≤ n”.

It is still the case that W(0) ⊃ W(1) is not a tautology, but now the formula that we create for W(2) (by another round of back-substitution) is “%o1 ≤ n”. (The variables %o1 and n are not modified in the loop body.) We now have that W(0) ∧ W(1) implies W(2).

By this means, the loop invariant synthesized for line 6 is “%g3 < n ∧ %o1 ≤ n”. This invariant implies that “%g3 < n” holds at line 6, which in turn implies that “%g2 < 4n” holds at line 7.

5.2.3 Discussion

In this section, we address the scalability of the verification phase, and discuss other potential improvements to the induction-iteration method.

One major cost of the verification phase is from performing the induction-iteration method, whose cost is determined by the number of iterations that have to be performed before an invariant is identified. The cost of iteration step of the induction-iteration method is determined by the cost to perform VC generation and invoking the theorem prover. These costs are ultimately determined by the characteristics of the untrusted code. From our experience, it seems to be sufficient to set the maximum allowable number of iterations to three. The intuition behind this number is as follows: the first iteration will incorporate the conditionals in the loop into L(1), the second iteration will test if L(1) is already a loop invariant, and no new information will be discovered beyond the second iteration. The situation for the inner loops is better when synthesizing loop invariants for an outer loop, since usually the tests in the inner loops will not contribute to the proof of a condition of an outer loop.

Besides the enhancements that were described in the previous section, there are a few enhancements that can, in principle, be made to our existing prototype:

- The first is to cache in the theorem prover: we can represent formulas in a canonical form and use previous results whenever possible.
- The second is to do tabulation at function calls or at nodes that have multiple incoming edges, and to reuse previous results of VCgen.
- The third is to use more efficient algorithms for simple formulas. Several people have described methods that trade the generality of the constraint system for better efficiency. Bodik *at al* [4] describe a method to eliminate array-bounds checks for Java programs. Their method uses a restricted form of linear constraints called *difference constraints* that can be solved using an efficient graph-traversal algorithm on demand. Wagner *et al* [53] have formulated the buffer overrun detection problem as *an integer constraint problem* that can be solved in linear time in practice.
- Finally, it might be profitable to use other invariant-synthesis methods in conjunction with induction iteration.

We have used the induction-iteration method to synthesize loop invariants because it works well for linear constraints and is totally mechanical. It is conceivable that we could use other techniques, such as the heuristic methods introduced by Katz and Manna [16] and Wegbreit [54], the difference equations method introduced by Elspas *et al* [10], and abstract interpretation using convex hulls [7].

The method described in [7] works forward on a program’s control-flow graph. It has the potential to speed up the induction-iteration method by pushing the facts down in the program’s control-flow graph. Simple experiments that we carried out demonstrated substantial speedups in the induction-iteration method by

		Sum	Paging Policy	Start Timer	Hash	Bubble Sort	Stop Timer	Btree	Btree2	Heap Sort 2	Heap Sort	jPVM	Stack-smashing	MD5
Number of Each Feature	**Instructions**	13	20	22	25	25	36	41	51	71	95	157	309	883
	Branches	2	5	1	4	5	3	11	11	9	16	12	89	11
	Loops (Inner loops)	1	2 (1)	0	1	2 (1)	0	2 (1)	2 (1)	4 (2)	4 (2)	3	7(1)	5(2)
	Procedure Calls	0	0	1	1	0	2	0	4	3	0	21(21)	2	6
	Global Safety Conditions	4	9	13	14	19	17	41	42	56	84	57	162	135
Time (Seconds)	**Typestate Propagation**	0.01	0.06	0.02	0.04	0.03	0.04	0.08	0.11	0.12	0.08	1.04	1.42	6.82
	Annotation + Local Verification	0.001	0.003	0.004	0.004	0.002	0.005	0.007	0.009	0.010	0.010	0.032	0.031	0.087
	Global Verification	0.05	0.41	0.06	0.35	0.45	0.08	0.50	0.41	2.05	3.58	4.18	10.15	7.04
	TOTAL	0.06	0.47	0.08	0.39	0.48	0.13	0.59	0.53	2.18	3.67	5.25	11.60	13.95

Figure 9: Characteristics of the Examples and Performance Results

selectively pushing conditions involving array bounds down in the program's control-flow graph. The abstract-interpretation method also addresses some of our current limitations such as inferring bounds of local arrays or arrays in structures (see Section 6).

6 Initial Experience

We have implemented a prototype safety checker for SPARC machine-language programs, and applied our safety checker to several examples. The time to check these examples varies from 0.06 seconds to 14 seconds.

The examples include array sum, start-timer and stop-timer code taken from Paradyn's performance-instrumentation suite [22], two versions of Btree traversal (one version compares keys via a function call), hash-table lookup, a kernel extension that implements a page-replacement policy [46], bubble sort, two versions of heap sort (one manually inlined version and one interprocedural version), stack-smashing (example 9.b described in Smith's paper [43]), `MD5Update` of the MD5 Message-Digest Algorithm [39], and `Java_jPVM_addhosts` of jPVM [15]. jPVM is a Java native interface to PVM for the Java platform. Java Native Interface (JNI) is a native-programming interface that allows Java code that runs inside a Java Virtual Machine to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly [14]. In the jPVM example, we verify that calls into JNI methods and PVM library functions are safe, i.e., they obey the safety preconditions (see Section 2). All examples are written in C and then compiled with `gcc -O` (version 2.7.2.3).

In our experiments, we were able to find a safety violation in the example that implements a page-replacement policy—it attempts to dereference a pointer that could be null—and we identified all array out-of-bounds violations in the stack-smashing example. Figure 9 summarizes the time needed to verify each of the examples on a 440MHz Sun Ultra 10 machine. The times are divided into the times to perform typestate propagation, create annotations and perform local verification, and perform global verification. It also characterizes the examples in terms of the number of machine instructions, number of branches, number of loops (total versus number of inner loops), number of calls (total versus number of calls to trusted functions),[3] and number of global safety conditions. The time to check these examples ranges from about 0.06 seconds to 14 seconds.

Our current approach has three major limitations. First, if the untrusted code uses local arrays, we may not be able to infer their bounds. For example, for the stack-smashing and MD5Update programs, we have to annotate the stackframes for the functions that use local arrays. Similarly, for structures that have multiple array-typed fields, our analysis may not be able to find out which array a pointer points to. To address this limitation, a forward pass such as that described in Cousot and Halbwachs [7] could be used to propagate the preconditions forward to find out information about the bounds of local arrays and to disambiguate pointers that point to arrays in structures.

Second, our analysis may lose precision due to array references. Recall that we use a single abstract location to summarize all elements of an array, and model a pointer to the array base (or to an arbitrary array element) as a pointer that may point to the summary location. Our analysis loses precision when we cannot determine whether an assignment kills all elements of an array. For example, our analysis reported that some actual parameters to the host methods and functions are undefined in the jPVM example, when they were in fact defined. We believe that dependence-analysis techniques such as those used in parallelizing compilers can be used to address this limitation.

Third, the type system described in this paper is too restrictive in that it does not incorporate a notion of subtyping for structures and pointers. This can cause the analysis to lose precision when certifying programs written in an object-oriented style.

Our experience to date allows us to make the following observations:

- Certain compiler optimizations, such as loop-invariant code motion and improved register-allocation algorithms, actually make the task of safety checking *easier.* The memory-usage analysis that is part of typestate checking can lose precision at instructions that access memory (rather than registers). When a better job of register allocation has been done, more precise

3. A trusted function is either a host function or a function that we trust. We check that calls to a trusted function obey the corresponding safety preconditions of the function.

typestate information will be recovered. Loop-invariant code motion makes induction iteration more efficient by making the loops smaller and simpler.

- Certain compiler optimizations, such as strength reduction and optimizations to address-calculations, *complicate* the task of global verification because they hide relationships that can be used by the induction-iteration method. Standard techniques should allow us to overcome this limitation [7,29].
- There are several strategies that makes the induction-iteration method more effective: First, because certain conditions in a loop can pollute L(*j*), instead of using wlp(loop-body, W(*i-1*)) as W(*i*), we compute the disjunctive normal form of wlp(loop-body, W(*i-1*)), and try each of its disjuncts as W(*i*) in turn. Second, we rank the potential candidates according to a simple heuristic, and test each candidate for W(*i*) using a breadth-first strategy, rather than a depth-first one. Finally, forward propagation of information about array bounds can substantially reduce the time spent in the induction-iteration method (it reduces the time needed to verify that a W(*i*) is true on entry, and it eliminates the need to use generalization to synthesize a loop invariant).
- Verifying an interprocedural version of an untrusted program can take less time than verifying a manually inlined version because the manually inlined version replicates the callee functions and the global conditions in the callee functions. This is a place where our analysis benefits from the procedure abstraction.

7 Related Work

There are several projects investigating topics related to our safety-checking technique. The approaches taken in these projects range from statically identifying common programming errors, to statically ensuring type safety, to the run-time checking of certain security properties.

The projects that are closest to ours are Proof-Carrying Code (PCC) [32], the Touchstone Certifying Compiler [31] and Typed-Assembly Language (TAL) [25,26,28]. Proof-Carrying Code (PCC) has a code producer generate not only the code but also a proof that the code is safe. Consequently, verification of the safety of untrusted code can be carried out by a small proof checker. Since manual generation of proofs is tedious and error-prone, a certifying compiler automates the generation of PCC by having the compiler generate code that carries proofs. Touchstone is a prototype certifying compiler that compiles a safe subset of C into machine code that carries proofs of type safety.

Morrisett *et al* [25,26,28] introduced the notion of typed assembly language (TAL). In their approach, type information from a high-level program is incorporated into the representation of the program in a platform-independent typed intermediate form, and carried through a series of transformations down to the level of the target code. The compiler can use the type information to perform sophisticated optimizations. Certain internal errors of a compiler can be detected by invoking a type-checker after each code transformation. A compiler that uses typed assembly language certifies type safety by ensuring that a well-typed source program always maps to a well-typed assembly program.

The most prominent difference between our approach and the certifying compiler (or the TAL) approach is a philosophical one. The certifying compiler approach enforces safety by preventing "bad" things from being expressible in a source language. For example, both the safe subset of C of the Touchstone compiler and the Popcorn language for TALx86 [28] do not allow pointer arithmetic, pointer casting, or explicit deallocation of memory. In contrast, we believe that safe code can be written in any language and produced by any compiler, as long as nothing "bad" is said in the code.

This philosophical difference has several implications. It gives the code producer the freedom to choose any language (including even "unsafe" languages such as C or assembly), and the freedom to produce the code with an off-the-shelf compiler or manually. It eliminates the dependence of safety on the correctness of a compiler. As with PCC, our technique checks the safety of the final product of the compiler. It leads to the decoupling of the safety policy from the source language, which in turn, makes it possible for safety checking to be performed with respect to an extensible set of safety properties that are specified on the host side.

The second important difference between our approach and the certifying compiler (or TAL) approach is that the safety properties we enforce are based on the notation of typestate, which provides information at a finer granularity than types.

Finally, neither Touchstone nor the Popcorn compiler track aliasing information. We have introduced an abstract storage model and extended typestate checking to also track pointers. As a result, the analysis we provide is more precise than that used in Popcorn and Touchstone.

In addition to these high-level differences, there are a few technical differences. It should be noted that our safety checker can be viewed as a certifier that generates proofs by first recovering type information that (may have) existed in the source-language program (an embodiment of a suggestion made by Necula and Lee [31, p. 342].) The approach used in our safety checker differs from that used in the Touchstone compiler in the following respects: First, Touchstone replaces the standard method for generating VCs, in which formulas are pushed backwards through the program, with a forward pass over the program that combines VC generation with symbolic execution. In contrast, our system uses a forward phase of typestate checking (which is a kind of symbolic execution) followed by a fairly standard backward phase of VC generation. The VC-generation phase is a backwards pass over the program for the usual reason; the advantage of propagating information backwards is that it avoids the existential quantifiers that arise when formulas are pushed in the forward direction to generate strongest postconditions; in a forward VC-generation phase, quantifiers accumulate—forcing one to work with larger and larger formulas. Second, our safety-checking analysis mechanically synthesizes loop invariants for bounds checking and alignment checking, whereas Touchstone generates code that contains explicit bounds checks and then removes those checks that it can prove to be redundant.

Comparing with TAL, the type system of TAL is richer in the sense that they model several language features that we have not considered so far, including exceptions, type variables, and existential types. However, their type system does not support general pointers into the stack; nor can stack and heap pointers be unified so that a function taking a tuple argument can be passed either a heap-allocated or stack-allocated tuple [27]. Furthermore, TALx86 introduce special macros for array subscripting and updating to prevent an optimizer from rescheduling them. The macros expand into code sequences that perform bound checks. We impose no such restrictions. TAL is more restrictive than PCC. PCC suggests that the relevant operational content of simple type systems may be encoded using extensions to first-order predicate logic. Our type system is closer in spirit to PCC, in that we provide a meta language to describe types including size and alignment constraints. Moreover, TAL achieve flow-sensitivity in a different way than we do; they label different blocks of code as different functions, and assign types to the registers at the level of basic blocks. We achieve flow-sensitivity through more traditional dataflow-analysis techniques. In this way we can use results from the pointer-analysis community in a more straightforward way. Despite the differences, it is interesting to note that if our safety checker were to be given programs written in typed assembly language rather than in an untyped machine language, less work would be required to recover type information and to perform overload resolution (although we would still have to propagate state and access information). This also applies to Java bytecode [19], where type information is contained in the bytecode instructions themselves.

A static debugger uses static analysis to find unsafe operations rather than to guarantee safety. Detlefs *et al* [8] describe a static checker for common programming errors, such as array index out-of-bounds, null-pointer dereferencing, and synchronization errors (in multi-threaded programs). In common with our approach, their analysis makes use of linear constraints, automatically synthesizes loop-invariants to perform bounds checking, and is parameterized by a policy specification. However, their safety-checking analysis works on source-language programs and also makes use of analyses that are neither sound nor complete. In their policy specifications, user-supplied `MODIFIES` lists (specifying which variables of a procedure can be modified) offer a certain degree of access control; our access policies are given in terms of regions, categories, and access permissions, which is a more general mechanism.

Leroy and Rouaix [18] have proposed a theoretical model for systematically placing type-based run-time checks into interface routines of the host code. Their technique differs from ours in several respects: it is dynamic, it checks the host rather than the untrusted code, and it requires that the source of the host API be available. Furthermore, safety requirements are specified by enumerating a set of predetermined sensitive locations and invariants on these locations, whereas our model of a safety policy is more general. Finally, they perform type checking, whereas we perform typestate checking.

8 Limitations

The main limitation of our technique is that it only can enforce safety properties that are expressible using typestates and linear constraints. This excludes all liveness properties and some safety properties.

Our analysis uses flow-sensitive interprocedural analysis to propagate typestate information. The verification phase is fairly costly due to the need to synthesize loop invariants to prove the safety predicates. The scalability of our analysis remains to be evaluated with bigger applications.

Like all static techniques, our technique is incomplete. First, the analysis loses precision when handling array references, because we use a single abstract location to summarize all elements of the array. Second, the existing prototype cannot infer the bounds of local arrays. We have to annotate the stackframes of functions that use local arrays. Third, the induction-iteration method itself is incomplete even for linear constraints. The induction-iteration method cannot prove the correctness of array accesses in a loop if correctness depends on some data whose values are set before the execution of the loop. One such example is the use of a sentinel at the end of the array to speed up a sequential search [49]. The generalization capabilities of the system may fall short for many problems, even though we only care about memory safety: The induction-iteration method could fail in cases that a loop invariant must be strengthened to the point where we end up verifying a large part of the partial correctness of the algorithm. Fourth, our type system does not incorporate a notion of subtyping for structures and pointers. This can hurt us when certifying programs written in an object-oriented style. In principle, we could define the meet operation for structures in terms of physical subtypes [42] (i.e., if $t_1 \leq t_2$ then struct t_2 is an initial prefix of struct t_1) and define the meet of two pointer types t_1 `ptr` and t_2 `ptr` to be ($t_1 \sqcap t_2$) `ptr`; however, the typestate-checking algorithm would have to be extended to track which pointers are mutable. (See [1] for a discussion of subtyping in the presence of mutable pointers.) Finally, our analysis is not able to deal with certain unconventional usages of operations, such as swapping two non-integer values by means of “exclusive or” operations.

Despite these limitations, the method shows some promise. Its limitations represent potential research opportunities, and we believe that future research will make the analysis more precise and efficient, and continued engineering can make the technique practical for larger programs.

Acknowledgments

We thank Rastislav Bodik, Jianjun Chen, Brian Wylie, Ari Tamches, and Philip Roth for their feedback on the paper. We thank the anonymous referees, whose comments have substantially improved both the technical quality and readability of the paper.

References

[1] M. Abadi, and L. Cardelli. **A Theory of Objects**. Monographs in Computer Science, D. Gries, and F. B. Schneider (Ed.). Springer-Verlag New York. (1996).

[2] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiucynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety, and Performance in the SPIN Operating System, *15th Symposium on Operating System Principles*. Copper Mountain, CO. (December 1995).

[3] B. Alpern, and F. B. Schneider. Recognizing Safety and Liveness. *Distributed Computing* **2** 3. (1987).

[4] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. *SIGPLAN Conference on Programming Language Design and Implementation.* Vancouver B.C., Canada. (June 2000).

[5] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of Pointers and Structures. *SIGPLAN Conference on Programming Language Design and Implementation.* New York, NY. (1990).

[6] T. Chiueh, G. Venkitachalam, P. Pradhan. Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions. 17th ACM Symposium on Operating Systems Principles. Charleston, SC. (December 1999).

[7] P. Cousot, and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. *Fifth Annual ACM Symposium on Principles of Programming Languages*. Tucson, AZ. (January 1978).

[8] D. K. Detlefs, R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. *Research Report 159*, Compaq Systems Research Center. Palo Alto, CA. (December 1998).

[9] E. W. Dijkstra. **A Discipline of Programming**. Prentice-Hall. Englewood Cliffs, NJ. (1976).

[10] B. Elspas, M. W. Green, K. N. Levitt, and R. J. Waldinger. **Research in Interactive Program-Proving Techniques**. SRI, Menlo Park, California. (May 1972).

[11] D. Engler, M. F. Kaashoek, and J. O’Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. *15th Symposium on Operating System Principles*. Copper Mountain, CO. (December 1995).

[12] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM* **12** 10. (October 1969).

[13] Illustra Information Technologies. **Illustra DataBlade Developer's Kit Architecture Manual,** Release 1.1. (1994).

[14] JavaSoft. **Java Native Interface Specification**. Release 1.1 (May 1997).

[15] jPVM: A Native Methods Interface to PVM for the Java Platform. http://www.chmsr.gatech.edu/jPVM. (2000).

[16] S. Katz, and Z. Manna. A Heuristic Approach to Program Verification. *3rd International Conference on Artificial Intelligence.* (August 1973).

[17] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnocott. The Omega Library, Version 1.1.0 Interface Guide. omega@cs.umd.edu. http://www.cs.umd.edu/projects/omega. (November 1996).

[18] X. Leroy, and F. Rouaix. Security Properties of Typed Applets. *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, CA. (January 1998).

[19] Lindholm T., and F. Yellin. **The Java (TM) Virtual Machine Specification.** Second Edition. http://java.sun.com/docs/books/vmspec/2ndedition/html/VMSpecToC.doc.html. (1999).

[20] S. McCanne, and V. Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. *The Winter 1993 USENIX Conference*. USENIX Association. San Diego, CA. (January 1993).

[21] Misrosoft. Microsoft COM Technologies-Information and Resources for the Component Object Model-Based Technologies. http://www.microsoft.com/com. (March 2000)

[22] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall.The Paradyn Parallel Performance Measurement Tools. *IEEE Computer* **28** 11. (November 1995).

[23] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The Packet Filter: An Efficient Mechanism for User-Level Network Code. *ACM Symposium on Operating Systems Principles.* Austin, TX. (November 1987).

[24] J. Morris. A General Axiom of Assignment. **Theoretical Foundations of Programming Methodology, Lecture Notes of an International Summer School, directed by F. L. Bauer, E. W. Dijkstra and C.A.R. Hoare**. Manfred Broy and Gunther Schmidt (Ed.). D. Reidel Publishing Company. (1982).

[25] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML Compiler: Performance and Safety Through Types. *In 1996 Workshop on Compiler Support for Systems Software.* Tucson, AZ. (February 1996).

[26] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *25th Annual ACM Symposium on Principles of Programming Languages*. San Diego, CA. (January 1998).

[27] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-Based Typed Assembly Language. *1998 Workshop on Types in Compilation*. Published in Xavier Leroy and Atsushi Ohori, editors, Lecture Notes in Computer Science, 1473. Springer-Verlag 1998.

[28] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, S. Zdancewic. TALx86: A Realistic Typed Assembly Language. ACM Workshop on Compiler Support for System Software. Atlanda, GA (May 1999).

[29] S. S. Muchnick. **Advanced Compiler Design and Implementation**. Morgan Kaufmann Publishers, Inc. (1997).

[30] G. Necula. Compiling with Proofs. *Ph.D. Dissertation,* Carnegie Mellon University. (September 1998).

[31] G. Necula, and P. Lee. The Design and Implementation of a Certifying Compiler. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. Montreal, Canada. (June 1998).

[32] G. Necula. Proof-Carrying Code. *24th Annual ACM Symposium on Principles of Programming Languages.* Paris, France. (January 1997).

[33] Netscape. Browser Plug-ins, 1999. http://home.netscape.com/plugins/index.html.

[34] C. Pu, T. Audrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. *15th ACM Symposium on Operating Systems Principles.* Copper Mountain, CO. (December 1995).

[35] W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Supercomputing*. Albuquerque, NM. (November 1991).

[36] W. Pugh, and D. Wonnacott. Eliminating False Data Dependences Using the Omega Test. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Francisco, CA. (June 1992).

[37] W. Pugh, and D. Wonnacott. Experience with Constraint-Based Array Dependence Analysis. *Technical Report CS-TR-3371.* University of Maryland. (1994).

[38] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM* **23** 2. (February 1980).

[39] R. Rivest. The MD5 Message-Digest Algorithm. **Request for Comments: 1321**. MIT Laboratory for Computer Science and RSA Data Security, Inc. (April 1992).

[40] F. B. Schneider. Towards Fault-Tolerant and Secure Agentry. *11th International Workshop on Distributed Algorithms.* Saarbrücken, Germany. (September 1997).

[41] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. *2nd USENIX Symposium on Operating Systems Design and Implementation*. Seattle, WA. (October 1996).

[42] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. *Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Toulouse, France. (September 1999).

[43] N. P. Smith. Stack Smashing Vulnerabilities in the UNIX Operating System. http://www.destroy.net/machines/security. (2000).

[44] R. Strom, and D. M. Yellin. Extending Typestate Checking Using Conditional Liveness Analysis. *IEEE Transactions on Software Engineering* **19** 5. (May 1993).

[45] R. Strom, and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering* **12** 1. (January 1986).

[46] C. Small, and M. A. Seltzer. Comparison of OS Extension Technologies. *USENIX 1996 Annual Technical Conference*. San Diego, CA. (January 1996).

[47] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. **Readings in Database Systems**. Second Edition. Michael Stonebraker (Ed.). (1994).

[48] Sun Microsystems, Inc. Java (TM) Plug-in Overview. http://java.sun.com/products/1.1.1/index-1.1.1.html. (October 1999).

[49] N. Susuki, and K. Ishihata. Implementation of an Array Bound Checker. *4th ACM Symposium on Principles of Programming Languages*. Los Angeles, CA. (January 1977).

[50] A. Tamches, and B. P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. *Third Symposium on Operating System Design and Implementation.* New Orleans, LA. *(*February 1999).

[51] M. Tamir. ADI: Automatic Derivation of Invariants. *IEEE Transactions on Software Engineering* **SE-6** 1. (January 1980).

[52] D. Tennenhouse, and D. Wetherall. Towards an Active Network Architecture. *Computer Communication Review* **26** 2. (April 1996).

[53] D. Wegner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. *The 2000 Network and Distributed Systems Security Conference.* San Diego, CA. (February 2000).

[54] B. Wegbreit. The Synthesis of Loop Predicates. *Communications of the ACM* **17** 2. (February 1974).

[55] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. *14th Symposium on Operating System Principles*. Asheville, NC. (December 1993).