Safety Checking of Machine Code

Zhichen Xu, Barton Miller, and Thomas Reps
Computer Sciences Department
University of Wisconsin at Madison
1210 W. Dayton Street
Madison, WI 53706
{zhichen,bart,reps}@cs.wisc.edu

1 INTRODUCTION

Importing and executing *untrusted foreign code* has become an everyday occurrence: Web servers download plug-ins and applets [22,34]; databases load type-specific extensions [8,33]; and operating systems load customized policies [1,6,15,23,27,29,36] and performance-measurement code [36]. There are even proposals for loading application-specific policies into Internet routers [38]. Certification of the safety of untrusted code is crucial in these domains.

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 a fixed safety policy for a specific source language (e.g., safe languages, certifying compilers [20], and typed-assembly languages [16,17,37]), our safety-checking technique (i) operates directly on binary code, independent of the compiler or language, and does not require that source code be available; (ii) 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) enforces not only 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, but also provides 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 typestates of 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. 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 that is specific to an individual program point. Typestate checking [30,31] differs from type checking in that type checking is a flow-*insensitive* analysis, whereas typestate checking is a flow-*sensitive* analysis.

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 [35] to synthesize loop invariants.

The main contributions of this paper are as follows:

- Our technique can be used to certify object code produced by an off-the-shelf compiler (independent of both the

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.

source language and the compiler). We require only that the inputs to the untrusted code be annotated with typestate information and linear constraints (i.e., linear equalities and inequalities that are combined with $\wedge$, $\vee$, $\neg$, and the quantifiers $\exists$ and $\forall$).

- The technique is extensible: 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.
- We extend the notion of typestate in several ways: (1) we use typestates to describe the state information of abstract locations in an abstract storage model; (2) we extend typestates to include access permissions (which are used to specify the extent to which untrusted code is allowed to access host resources); (3) 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 [5,7]. 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 simple examples (ranging from code that contains just a few branches, to code that contains nested loops). The safety checker was able to mechanically synthesize the loop invariants in all cases, and was able to prove that these examples met the necessary safety conditions, in times ranging from less than a second to about 6 seconds on an UltraSPARC machine (see Section 6). We have 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 paper presents an intraprocedural version of our analysis. The interprocedural version of the analysis will be discussed in the full paper.

The remainder of the paper is organized as follows: Section 2 describes the safety properties that we enforce and the notion of a safety policy. Section 3 gives an overview of the safety-checking analysis. Section 4 describes the basis of our 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.

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,* and *x*, meaning readable, writable, followable (i.e., can be dereferenced), and executable, respectively. In our model, *r*, *f*, and *x* are properties of the value stored in a location, whereas *w* is a property of the location itself. [2]

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 only the `tid` and `lwpid` fields, and to read and 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` : *r*], [*H* : `thread.next` : *rf*]

The above model can be used to specify a variety of different safety policies. For example, we can specify something roughly equivalent to *sandboxing* [39]. 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 read and write 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 data 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.

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

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

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.

Untrusted Code	Host Typestate	Safety Policy	Invocation
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	e: <int, initialized, r> arr: <int [n], {e}, rf> {n≥1} *arr is an integer array of size* n, *where* n≥1. e *is an abstract location that summarizes all elements of* arr.	V = {e, arr} [V : int : r] [V : int [n] : rf] *arr and* e *are in the V region. All integers in the V region are read-only. All base addresses to an integer array of size* n *in the V region are readable and followable.*	%o0 ← arr %o1 ← n *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

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 read-only, and that all base addresses to an integer array of size n in the V region are readable 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.

Initial Annotations	
Initial Typestate	**Initial Constraints**
e:<int, initialized, r> %o0:<int [n], {e}, rwf> %o1:<int, initialized, rw>	n ≥ 1 ∧ n=%o1

Figure 2: Initial Annotations

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 the linear constraint "n=%o1". Note that %o0 and %o1 both have the *w* access permission. This refers to the registers themselves (i.e., the untrusted code is permitted to 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, is *r* (and not *rw*).

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 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.

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	*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 4: *Local verification* checks the local safety preconditions. (In our example, the local safety preconditions are all true at 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 an optional attribute *w* to indicate whether the location is writable. 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 *name*(*l*), *size*(*l*), *align*(*l*), to denote the name, size, and alignment, respectively, of abstract location *l*.

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. For types, the meet of two different non-pointer types, or the meet of a pointer and non-pointer type is $\bot_t$, whereas the meet of two pointer types t_1 `ptr` and t_2 `ptr` is $(t_1 \sqcap t_2)$`ptr`.

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 $\bot_s$ that denotes an undefined value of any type. Figure 4 illustrates selected elements of the state lattice.

An access permission is either a subset of $\{r, f, x\}$, 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 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.

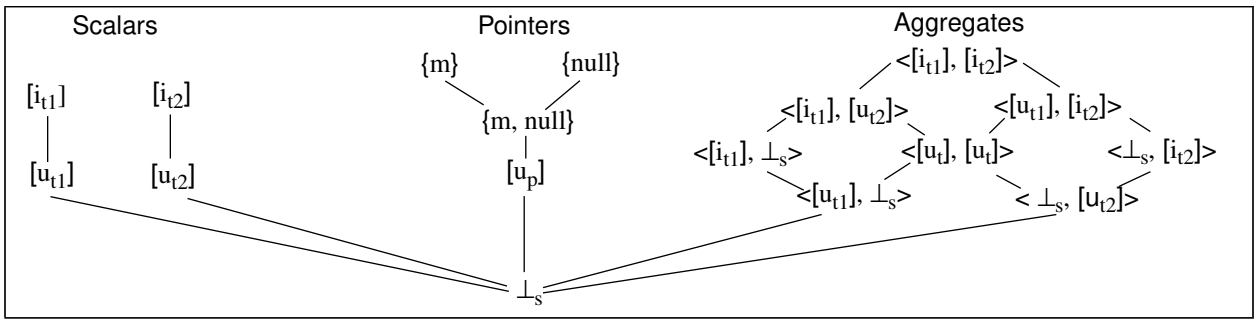

Figure 4: A Portion of the State Lattice

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 references to abstract locations, where one of the elements of P can be null. *For reference 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.*

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

The typestate lattice also includes a top element, denoted by $\top$.

4.2 An Abstract Operational Semantics for SPARC Machine Instructions

An abstract store is given by a total map $\mathcal{M}$: *absLoc* $\rightarrow$ typestate. We define the abstract operational semantics of a SPARC machine instruction as a transition function $\mathcal{R}$: $\mathcal{M} \rightarrow \mathcal{M}$. 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 $\top$. Consequently, during typestate checking, propagation of information through the instructions of a loop is delayed until a non-$\top$ 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) rather than type checking (which is flow insensitive). 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.

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 $\varnothing$ if no such fields exist.

1. The typestate-propagation rules for scalar-add state that after the execution of the `add` instruction, the type and state of $\mathbf{r}_d$ is the meet of those of $\mathbf{r}_s$ and *Opnd* before the execution, and the type and state of all other abstract locations in *absLoc* remain unchanged.
2. For an array-index calculation, the type of the destination register becomes "*t* `ptr`", where "*t*" is the type of an array element. 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. When $|F|=1$, "$\mathbf{r}_a$+*n*" must point to the destination, so the destination receives the typestate of the source register; When $|F|>1$, all we know is that "$\mathbf{r}_a$+*n*" may point to the locations in *F*; hence each possible destination receives the meet of the typestate before the operation and the typestate of the source register.

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. $A'(\mathbf{r}_d) = A(\mathbf{r}_s)$ 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 `ptr` 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 $\mid F \mid = 1$, $T'(l)=T(\mathbf{r}_s)$, where $F=\{l\}$ 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 $\mid F \mid=1$, $S'(l)=S(\mathbf{r}_s)$, where $F=\{l\}$ 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 $\mid F \mid=1$, $A'(l) = A(\mathbf{r}_s)$, where $F=\{l\}$ 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

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* $r \in A(l)$ 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 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), S(l) $\notin$ {[$u_{T(l)}$], $\perp_s$}, and T(l) is a pointer type; the predicate *assignable*(m, l) is true *iff readable*(m), *writable*(l), and (T(l)=T(m) or (*align*(l) mod *align*(T(m))=0 and *size*(T(m))≤*size*(l))) all hold; the predicate *align*(*address, n*) is true *iff address* mod n = 0.

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

Table 2: Attachment of Safety Properties

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 assignment of ⊤ to all abstract locations 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 <$\perp_t$, $\perp_s$, ∅>.

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

The Appendix shows the results of typestate propagation applied to our running example. In particular, it shows that, for the `ld` instruction at line 7 in Figure 1, `%o2` holds the base address of an integer array and `%g2` is an integer (and hence is 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 [10]. 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 $\neg$, $\wedge$, and $\vee$, and the quantifiers $\forall$ and $\exists$. The affine constraints can be either equality or inequality constraints [10]. Presburger formulas are decidable. For more details on the Omega Library, see Kelly *et al* [10] and Pugh *et al* [24,25,26].

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 [35]. We have extended the induction-iteration method to synthesize loop invariants for natural loops (and also to work on machine-language programs).

The induction-iteration method uses the "weakest liberal precondition" (wlp) as a heuristic for generating loop invariants. It attempts to find a loop invariant of the form $\bigwedge_{j \geq i \geq 0} W(i)$, where W(*0*) is the wlp generated by back-substituting the condition to be proved until the entry of a loop is reached; W(*i+1*) is computed by back-substituting W(*i*) across the loop body until the loop entry is reached again. Suzuki and Ishihata [35] have shown that establishing that $\bigwedge_{j \geq i \geq 0} W(i)$ is a loop invariant is equivalent to showing that (i) each W(*i*), $j \geq i \geq 0$, is true on entry to the loop, and (ii) $(\bigwedge_{j \geq i \geq 0} W(i))$ implies W(*j+1*).

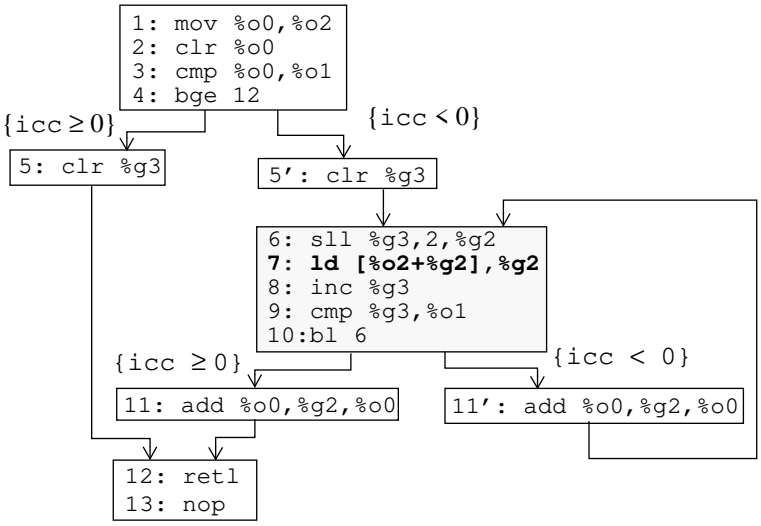

Figure 5: Induction Iteration: Example

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 5. 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.

To verify that "`%g2` < 4`n`" holds at line 7, we perform back-substitution, starting with "`%g2` < 4`n`". 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 a technique called generalization (introduced by Suzuki and Ishihata [35]). The generalization of a formula *F* is defined as ¬(eliminate(¬*F*)), where eliminate uses the Fourier-Motzkin variable-elimination method to eliminate variables from the set of variables in ¬*F,* thereby generating a simplified set of constraints that has the same integer solution as ¬*F*. If there are several resulting generalizations, each of them in turn is chosen as the candidate generalized formula.

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.

6 INITIAL EXPERIENCE

We have implemented a prototype safety checker for SPARC machine-language programs, and applied our safety checker to several simple examples. The examples include array sum, start-timer and stop-timer code taken from Paradyn's performance-instrumentation suite [14], 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 [32], bubble sort, and heap sort. All examples are written in C and then compiled with `gcc -O` (version 2.7.2.3).

Figure 6 summarizes the time needed to verify each of the examples on a 300MHz Sun Ultra 10 machine (broken down 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, number of calls, and number of global safety conditions. The time to check these examples ranges from about 0.1 seconds to 6 seconds.

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

- First, 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 carried out, more precise typestate information will be recovered. Loop-invariant code

motion makes induction iteration more efficient by making the loops smaller and simpler.

- Second, 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 [3,18]. We are currently implementing a scheme to recover the relationships among the induction variables. (This algorithm is based on the algorithm from [18] for identifying induction variables.)
- There are several strategies that are effective in speeding up the synthesis of loop invariants: (i) because certain conditions in a loop can pollute W(*i*) during induction iteration, instead of generalizing W(i) immediately, we compute the disjunctive normal form of W(*i*), and try each disjunct of W(*i*) in turn; (ii) instead of testing the disjuncts blindly in arbitrary order, we rank the candidates according to a simple heuristic criterion; (iii) we test the potential candidates for W(*i*) using a breadth-first strategy, rather than a depth-first one.

		Sum	Start Timer	Stop Timer	Btree	Btree2	Paging Policy	Hash	Bubble Sort	Heap Sort
Number of Each Feature	**Instructions**	13	22	36	41	51	20	25	25	95
	Branches	2	1	3	11	11	5	4	5	15
	Loops (Inner loops)	1	0	0	2 (1)	2 (1)	2 (1)	1	2 (1)	4 (2)
	Procedure Calls	0	1	2	0	4	0	1	0	0
	Global Safety Conditions	4	13	17	41	42	9	14	19	84
Time (Seconds)	**Typestate Propagation**	0.02	0.03	0.05	0.18	0.16	0.10	0.05	0.05	0.14
	Annotation + Local Verification	0.002	0.006	0.008	0.01	0.01	0.004	0.006	0.006	0.03
	Global Verification	0.07	0.09	0.12	0.69	0.62	0.55	1.51	1.21	5.93

Figure 6: Characteristics of the Examples and Performance Results

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.

Proof-Carrying Code (PCC) [21] 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 [20] 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.

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. (Thus our tool can be viewed as an embodiment of a suggestion made by Necula and Lee [20, p. 342].) However, the approach used in our safety checker differs from that used in the Touchstone compiler in the following respects: First, the safety properties that we enforce are based on the notion of typestates. Although in this paper we have only used typestates to distinguish initialized values from uninitialized ones, the notion of typestate is more general. For example, it is straightforward to use typestates to check the matching of resource allocations and deallocations in untrusted code. Second, 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. Third, unlike Touchstone, which implements a fixed safety policy, our safety checker is parameterized by a host-specified access policy. Finally, 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.

Morrisett and Tarditi [16,17,37] introduced the notion of typed assembly language. 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. Although a compiler that uses typed assembly language ensures that a well-typed source program always maps to a well-typed assembly program, it can only detect violations of properties that are enforced by the type system. In their work, the type system does not use access permissions to protect host resources, nor does it synthesize loop invariants for bounds checking and alignment checking. Although their type system does include a limited form of state information for uninitialized values, it is less general than the notion of typestate. However, 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 [12], 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* [4] 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 [11] 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.

REFERENCES

[1] 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).

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

[3] 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).

[4] 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).

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

[6] 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).

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

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

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

[10] 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).

[11] 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).

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

[13] 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).

[14] 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).

[15] 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).

[16] 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).

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

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

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

[20] 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).

[21] G. Necula, and P. Lee. Proof-Carrying Code. *Technical Report CMU-CS-96-165*. (September 1996).

[22] Netscape. Browser Plug-ins. http://home.netscape.com/plugins/index.html. (1999).

[23] 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).

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

[25] 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).

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

[27] 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).

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

[29] 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).

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

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

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

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

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

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

[36] 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).

[37] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-directed Optimizing Compiler for ML. *ACM Conference on Programming Language Design and Implementation.* Philadelphia, PA. (May 1996).

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

[39] 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).

APPENDIX

Figure 7 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).

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

Figure 7: Results of Typestate Propagation