Typestate Checking of Machine Code

Zhichen Xu, Thomas Reps, and Barton P. Miller
University of Wisconsin at Madison
{zhichen, reps, bart}@cs.wisc.edu

Abstract

We check statically whether it is safe for untrusted foreign machine code to be loaded into a trusted host system. (Here "safety" means that the program abides by a memory-access policy that is supplied on the host side.) Our technique works on ordinary machine code, and mechanically synthesizes (and verifies) a safety proof. Our earlier work along these lines was based on a C-like type system, which does not suffice for machine code whose origin is C++ source code. In the present paper, we address this limitation with an improved typestate system and introduce several new techniques, including: summarizing the effects of function calls so that our analysis can stop at trusted boundaries, inferring information about the sizes and types of stack-allocated arrays, and a symbolic range analysis for propagating information about array bounds. These techniques make our approach to safety checking more precise, more efficient, and able to handle a larger collection of real-life code sequences than was previously the case. For example, allowing subtyping among structures and pointers allowed our implementation to analyze code originating from object-oriented source code. The use of symbolic range analysis eliminated 60% of the total attempts to synthesize loop invariants in the 11 programs of our test suite that have array accesses. In 4 of these programs, it eliminated the need to synthesize loop invariants altogether. The resulting speedup for the global-verification phase of the system ranges from -0.4% to 63% (with a median of 37%).

1 Introduction

Our goal is to check statically whether it is safe for a piece of untrusted foreign machine code to be loaded into a trusted host system. We start with ordinary machine code and mechanically synthesize (and verify) a safety proof. In an earlier paper [35], we reported on initial results from our approach, the chief advantage of which is that it opens up the possibility of being able to certify code produced by a general-purpose off-the-shelf compiler from programs written in languages such as C, C++, and Fortran. Furthermore, in our work we do not limit the safety policy to just a fixed set of memory-access conditions that must be avoided; instead, we perform safety checking with respect to a safety policy that is supplied on the host side. Our earlier work was based on a C-like type system, which does not suffice for machine code whose origin is C++ source code. In the present paper, we address this limitation and also introduce several other techniques that make our safety-checking analysis more precise and scalable.

When our proof-synthesis techniques are employed on the host side, our approach can be viewed as an *alternative* to the Proof-Carrying Code (PCC) approach [20]: PCC requires a code producer to create not just the machine code but also a proof that the code is safe, and then has the host perform a proof-validation step. When our proof-synthesis techniques are employed by the code producer (on the foreign side of the untrusted/trusted boundary), our approach can be viewed as an *ally* of PCC that helps to lift current limitations of certifying compilers [7, 18, 19], which produce PCC automatically, but only for programs written in certain safe source languages.

In this paper, we present the following improved techniques for use in proof synthesis:

- An improved typestate-checking system that allows us to perform safety-checking on untrusted machine code that implements inheritance polymorphism via physical subtyping [24]. This work introduces a new method for coping with subtyping in the presence of mutable pointers (Section 3).
- A mechanism for summarizing the effects of function calls via safety pre- and post-conditions. These summaries allow our analysis to stop at trusted boundaries. They form a first step toward checking untrusted code in a modular fashion, which makes the safety-checking technique more scalable (Section 4).
- A technique to infer information about the sizes and types of stack-allocated arrays (local arrays). This was left as an open problem in our previous paper [35] (Section 5).

- A symbolic range analysis that is suitable for propagating information about array bounds. This analysis makes the safety-checking algorithm less dependent upon expensive program-verification techniques (Section 6).

Section 2 provides a brief review of the safety-checking technique from our earlier work [35]. Section 7 illustrates the benefits of our techniques via a few case studies. Section 8 compares our techniques with related work.

As a result of these improvements, we can handle a broader class of real-life code sequences with better precision and efficiency. For example, allowing subtyping among structures and pointers allows us to analyze code originating from object-oriented source code. The use of symbolic range analysis eliminated 60% of the total attempts to synthesize loop invariants in the 11 programs of our test suite that have array accesses. In 4 of these programs, it eliminated the need to synthesize loop invariants altogether. The resulting speedup for global verification ranges from -0.4% to 63% (with a median of 37%). Together with improvements that we made to our global-verification phase, range analysis allows us to verify untrusted code that we were not able to handle previously.

2 Safety Checking of Machine Code

In this section, we briefly review the safety-checking technique from our earlier work [35]. The safety-checking analysis enforces a default collection of safety conditions to prevent type violations, array out-of-bounds violations, address-alignment violations, uses of uninitialized variables, null-pointer dereferences. In addition, the host side can specify a precise and flexible *access policy*. This access policy specifies the host data that can be accessed by the untrusted code, and the host functions (methods) that can be called. It provides a means for the host to specify the "least privilege" the untrusted code needs to accomplish its task.

Our approach is based on annotating the global data in the host. The type information in the untrusted code is inferred. Our analysis starts with information about the initial memory state at the entry of the untrusted code. It abstractly interprets the untrusted code to produce a safe approximation of the memory state at each program point. It then annotates each instruction with the safety conditions each instruction must obey and checks these conditions.

The memory states at the entry, and other program points of the untrusted code, are described in terms of an *abstract storage model*. An *abstract store* is a total map from *abstract locations* to *typestates*. An abstract location summarizes one or more physical locations so that our analysis has a finite domain to work over. A typestate describes the type, state, and access permissions of the values stored in an abstract location.

The initial memory state at the entry of the untrusted code is given by a *host-typestate specification*, and an *invocation specification*. The host typestate specification describes the type and the state of the host data before the invocation of the untrusted code, as well as safety pre- and post-conditions for calling host functions (methods). The invocation specification provides the binding information from host resources to registers and memory locations that represent initial inputs to the untrusted code.

The safety-checking analysis consists of five phases. The first phase, *preparation,* combines the information that is provided by the host-typestate specification, the invocation specification, and the access policy to produce an initial annotation (in the form of an abstract store for the program's entry point). It also produces an interprocedural control-flow graph for the untrusted code. The second phase, *typestate-propagation,* takes the control-flow graph and the initial annotation as inputs. It abstractly interprets the untrusted code to produce a safe approximation of the memory contents (i.e., a typestate for each abstract location) at each program point. The third phase, *annotation*, takes as input the typestate information discovered in the typestate-propagation phase, and annotates each instruction with *local* and *global safety conditions* and *assertions*: the local safety preconditions are conditions that can be checked using typestate information alone; the assertions are restatements (as logical formulas) of facts that are implicit in the typestate information. The fourth phase, *local verification*, checks the local safety conditions. The fifth phase, *global verification,* verifies the global safety conditions using program-verification techniques [30].

At present, our implementation handles only non-recursive programs.

3 An Improved Typestate System

In our past work, our analysis made the assumption that a register or memory location stored values of a single type at any given program point (although a register/memory location could store different types of values at different program points). However, this approach had some drawbacks for programs written in languages that support subtyping and inheritance, and also for programs written in languages like C in which programmers have the ability to simulate subtyping and inheritance.

In this section, we describe how we have extended the typestate system [35] to incorporate a notion of subtyping among pointers. With this approach, each use of a register or memory location at a given occurrence of an instruction is resolved to a polymorphic type (i.e., a super type of the acceptable values). In the rest of this section, we describe the improved type component of our typestate system.

t :: ground	*Ground 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*
| $\top(n)$	*Top type of n bits*
| $\perp(n)$	*Bottom type of n bits (Type "any" of n bits)*
m:: (t, l, i)	*Member labeled l of type t at offset i*
ground:: `int`(g:s:v) | `uint`(g:s:v) | ...	

Figure 1: A Simple Language of Type Expressions.
t stands for type, and m stands for a struct or union member.
Although the language in which we have chosen to express the type system looks a bit like C, we do not assume that the untrusted code was necessarily written in C or C++.

3.1 Type Expressions

Figure 1 shows the language of type expressions used in the typestate system. Compared with our previous work, the typestate system now includes (i) bit-level representations of integer types, and (ii) top and bottom types that are parameterized with a size parameter. The type `int`(g:s:v) represents a signed integer that has $g+s+v$ bits, of which the highest g bits are ignored, the middle s bits represent the sign or are the result of a sign extension, and the lowest v bits represent the value. For example, a 32-bit signed integer is represented as `int`(0:1:31), and an 8-bit signed integer (e.g., a C/C++ `char`) with a 24-bit sign extension is represented as `int`(0:25:7). The type `uint`(g:s:v) represents an unsigned integer, whose middle s bits are zeros. The type $t\ (n]$ denotes a pointer that points somewhere into the middle of an array of type t of size n.

The bit-level representation of integers allows us to express the effect of instructions that load (or store) partial words. For example, the following code fragment (in SPARC machine language) copies a character pointed to by register `%o1` to the location that is pointed to by register `%o0`:

```
ldub [%o1],%g2
stb %g2,[%o0]
```

If `%o1` points to a signed character and a C-like type system is used (i.e., as in [35]), typestate checking will lose precision when checking the above code fragment. There is a loss of precision because the instruction "`ldub [%o1], %g2`" loads register `%g2` with a byte from memory and zero-fills the highest 24 bits, and thus the type system of [35] treats the value in `%g2` as an unsigned integer. In contrast, with the bit-level integer types of Figure 1, we can assign the type `int`(24:1:7) to `%g2` after the execution of the load instruction. This preserves the fact that the lowest 8 bits of `%g2` store a signed character (i.e., an `int`(0:1:7)).

$$[\mathit{Reflexivity}]\ \frac{}{t <: t} \qquad [\mathit{Top}]\ \frac{}{\top(\mathit{sizeof}(t)) <: t} \qquad [\mathit{Bottom}]\ \frac{}{t <: \bot(\mathit{sizeof}(t))}$$

$$[\mathit{Ground}]\ \frac{g+s+v=g'+s'+v',\ g \le g',\ v \le v'}{\begin{array}{l}\mathtt{int}(g{:}s{:}v) <: \mathtt{int}(g'{:}s'{:}v') \\ \mathtt{uint}(g{:}s{:}v) <: \mathtt{uint}(g'{:}s'{:}v') \\ \mathtt{uint}(g{:}s{:}v) <: \mathtt{int}(g'{:}s'{:}v')\end{array}} \qquad [\mathit{First\ Member}]\ \frac{m_1 = (l, t, 0),\ t <: t'}{s\{m_1, \ldots, m_k\} <: t'}$$

$$[\mathit{Structures}]\ \frac{k' \le k,\ m_1 <: m'_1, \ldots, m_{k'} <: m'_{k'}}{s(m_1, \ldots, m_k) <: s\,(m'_1, \ldots, m'_{k'})} \qquad [\mathit{Members}]\ \frac{m=(l, t, i),\ m'=(l', t', i'),\ l=l',\ i=i',\ t <: t'}{m <: m'}$$

$$[\mathit{Pointer}]\ \frac{t <: t'}{t\ \mathtt{ptr} <: t'\ \mathtt{ptr}} \qquad [\mathit{Array}]\ \frac{}{t\,[i] <: t\,(i)}$$

Figure 2: Inference Rules that Define the Subtyping Relation.

3.2 A Subtyping Relation

We now introduce a notion of subtyping on type expressions, adopted from the *physical-subtyping* system of [24], which takes into account the layout of aggregate fields in memory. Figure 2 lists the rules that define when a type *t* is a *physical subtype* of *t*' (denoted by $t <: t'$).[1] In Figure 3, the rules [*Top*], [*Bottom*], [*Ground*], [*Pointer*], and [*Array*] are our additions to the physical-subtyping system given in [24]. An integer type *t* is a subtype of type *t'* if the range represented by *t* is a subrange of the range represented by *t',* and *t* has at least as many sign-extension bits as *t'*. Rule [*First Member*] states that a structure is a subtype of a scalar type if the type of the first member of the structure is a subtype of the scalar type. The consequence of this rule is that it is valid for a program to pass a structure in a place where a supertype of its first member is expected. The rules [*Structures*] and [*Members*] state that a structure *s* is a subtype of *s'* if *s'* is a prefix of *s*, and each member of *s'* is a supertype of the corresponding member of *s.* Rule [*Members*] gives the constraints on the corresponding members of two structures. The rule [*Pointer*] states if *t* is a subtype of *t'*, than *t* `ptr` is a subtype of *t'* `ptr`. *Rule* [*Array*] states that a pointer to the base of an array is a subtype of a pointer into the middle of an array. In our system, an assignment is legal *only if* the type of the right-hand-side expression is a physical subtype of the type of the receiving location, and the receiving location has enough space. The Rule [*Array*] is valid because t (i) describes a larger set of states than t [i]. (The global-verification phase of the analysis will check that all array references are within bounds.)

1. Note that the subtype ordering is conventional. However, during typestate checking the ordering is flipped: $t_1 \le t_2$ in the type lattice *iff* $t_2 <: t_1$.

Allowing subtyping among integer types, structures, and pointers allows the analysis to handle code that implements inheritance polymorphism via physical subtyping. For example, for a function that accepts a 32-bit integer, it is legal to invoke the function with an actual parameter that is a signed character (i.e., `int(0:1:7)`), provided that the value of the actual parameter is stored into a register or into memory via an instruction that handles sign extension properly. Figure 3 shows another example that involves subtyping among structures and pointers. According to the subtyping inference rules for structures and pointers, type `ColorPointPtr` is a subtype of `PointPtr`. Function `f` is polymorphic because it is legal to pass an actual parameter that is of type `ColorPointPtr` to function `f`.

```
struct Point {
    int(0:1:31) x;
    int(0:1:31) y;
};
```

```
struct ColorPoint {
    int(0:1:31) x;
    int(0:1:31) y;
    uint(24:0:8) color;
};
```

```
void f(Point* p) {
    p->x++;
    p->y--;
}
```

Figure 3: Subtyping Among Pointer Types

For object-oriented languages such as C++, there is an additional complication that arises from the use of virtual functions, where a virtual function could be implemented by any of the subclasses. As long as we have full information about the class hierarchy, we can simply assume that the callee of a call to a virtual function can be any of the functions that implement the virtual function and check all of them.

3.3 The State and Access Component of our Typestate system

We briefly review the state and access components of the typestate system. The state lattice contains a bottom element $\perp_s$ that denotes an undefined value of any type. For a scalar type t, its state can be u or i, which denote uninitialized and initialized values, respectively. We define $u \le i$ in the state lattice. For a pointer type p, its state can be either u or P (a non-empty set of abstract locations referenced); we define $u \le P$. One of the elements of P can be `null`. For sets P_1 and P_2, we define $P_1 \le P_2$ *iff* $P_2 \subseteq P_1$. For an aggregate type G, its state is given by the states of its fields.

An access permission is either a subset of $\{f, x, o\}$, or a tuple of access permissions. 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 "examine", "copy", and perform other operations not covered by x and f. 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.

3.4 Type Checking with Subtyping vs. Typestate Checking with Subtyping

Readers who are familiar with the problems encountered with subtyping in the presence of mutable pointers may be suspicious of rule [*Pointer*]. In fact, rule [*Pointer*] is unsound for traditional flow-insensitive type systems in the absence of alias information. This is because a flow-insensitive analysis that does not account for aliasing is unable to determine whether there are any indirect modifications to a shared data structure, and some indirect modifications can have disastrous effects. Figure 4 provides a concrete example of this. The statement at line 8 changes `clrPtr` to point to an object of the type `Point` indirectly via the variable `t`, so that `clrPtr` can no longer fulfill the obligation to supply the `color` field at line 9.

A static technique to handle this problem has to be able to detect whether such disastrous indirect modifications could happen. There are several approaches to this problem found in the literature. For example, the linear type system given in [33] avoids aliases altogether (and hence any indirect modifications) by "consuming" a pointer as soon

```
typedef Point *PointPtr;
typedef ColorPoint *ColorPointPtr;

1: ColorPoint clr;
2: Point bw;
3: void f2(void) {
4:     PointPtr bwPtr = &bw;
5:     ColorPointPtr clrPtr = &clr;
6:     ColorPointPtr *r = &clrPtr;
7:     PointPtr *t = r;
8:     *t = bwPtr;
9:     clrPtr->color = 1;
10:}
```

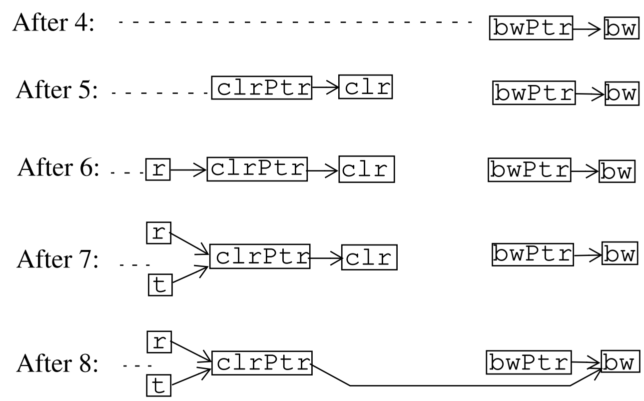


Figure 4: Rule [Pointer] is unsound for flow-insensitive type checking in the absence of aliasing information.
(*Assume the same type declarations as shown in Figure 3*.)

as it is used once. Smith *et al* [29] use singleton types to track pointers, and alias constraints to model the shape of the store. (Their goal is to tracks non-aliasing to facilitate memory reuse and safe deallocation of objects.)

Another approach involves introducing the notions of immutable fields and objects [1]. The idea is that if *t* is a subtype of type *t'*, type *t* `ptr` is a subtype of *t'* `ptr` only if any field of *t* that is a subtype of the corresponding field of *t'* is immutable. Moreover, if a field of *t* is a pointer, then the object pointed by it must also be immutable. This rule applies transitively. For this approach to work correctly, a mechanism is needed to enforce these immutability restrictions.

Our work represents yet a fourth technique. Our system performs typestate checking, which is a flow-sensitive analysis that tracks aliasing relationships among abstract locations. (These state descriptors resemble the storage-shape graphs of Chase *et al* [4], and are similar to the diagrams shown in the right-hand column of Figure 4.) By inspecting the storage-shape graphs at program points that access heap-allocated storage, we can (safely) detect whether an illegal field access can occur. For instance, from the shape graph that arises after statement 8 in Figure 4, the analysis can determine that the access to `color` in statement 9 represents a possible memory-access error. Programs with such accesses are rejected by our safety checker.

4 Summarizing Function Calls

By summarizing function calls, the safety-checking analysis can stop at the boundaries of trusted code. Instead of tracing into the body of a trusted callee, the analysis can check that a call obeys a safety pre-condition, and then use the post-condition in the rest of the analysis. We describe a method for summarizing trusted calls with safety pre- and post-conditions in terms of abstract locations, typestates, and linear constraints. The safety pre-conditions describe the obligations that the actual parameters must meet, whereas the post-conditions provide a guarantee on the resulting state.

Currently, we produce the safety pre- and post-conditions by hand. This process is error-prone, and it would be desirable to automate the generation of function summaries. Recent work on interprocedural pointer analysis has shown that pointer analysis can be performed in a modular fashion [5, 6]. These techniques analyze each function assuming unknown initial values for parameters (and globals) at a function's entry point to obtain a *summary function* for the dataflow effect of the function. In future work, we will investigate how to use such techniques to create safety pre- and post- conditions automatically.

We represent the obligation that must be provided by an actual parameter as a *placeholder* abstract location (placeholder) whose size, access permissions, and typestate provide the detailed requirements that the actual parameter must satisfy. When a formal parameter is a pointer, its state descriptor can include references to other placeholders that represent the obligations that must be provided by the locations that may be pointed to by the actual parameter. In our model, the state descriptor of a pointer-typed placeholder can refer to `null`, to a placeholder, or to a placeholder and `null`. If it refers to just `null`, then the actual parameter must point to `null`. If it refers to a placeholder, then all locations that are pointed to by the actual parameter must satisfy the obligation denoted by the placeholder. If the state descriptor refers to both `null` and a placeholder, then the actual parameter must either point to `null`, or to locations that satisfy the obligation. We represent the pre-conditions as a list of the form "*placeholder* : *typestate*".

The safety post-conditions provide a way for the safety-checking analysis to compute the resulting state of a call to a summarized function. They are represented by a list of post-conditions of the form [*alias context*, *placeholder* : *typestate*]. An *alias context* [5] is a set of potential aliases (*l* eq *l'*) (or potential non-aliases (*l* neq *l'*)), where *l* and *l'* are placeholders. The alias contexts capture how aliasing among the actual parameters can affect the resulting state.

The safety pre- and post-conditions can also include linear constraints. When they appear in the safety pre-conditions, they represent additional safety requirements. When they appear in the post-conditions, they provide additional information about the resulting memory state after the call.

To make this idea concrete, Figure 5 shows an example that summarizes the C library function `gettimeofday`. It specifies that for the call to be safe, `%o0` must either be (i) `null` or (ii) be the address of a writable location of size sufficient for storing a value of the type `struct timeval`.The safety post-conditions specify that after the execution of the call, the two fields of the location will be initialized, and `%o0` will be an initialized integer. (On SPARC, the actual parameters will be passed through the registers `%o0`, `%o1`, ..., `%o5`, and the return value of the function will be stored in the register `%o0`.)

```
int gettimeofday (struct timeval *tp);
    Safety Pre-condition:
        %o0: <struct timeval ptr, {null, t}, fo>
        t: <struct timeval, u, wo>
    Safety Post-condition:
        [(), t: <struct timeval, [0:<int(0:1:31), i, o>, 32:<int(0:1:31), i, o>], o>]
        [(), %o0 : <int(0:1:31), i, o>]
        [(), %o1-%o5, %g1-%g7: <⊥(32), ⊥, o>]
```

Figure 5: Safety Pre- and Post- Conditions.
The typestate of an aggregate is given by the typestates of its components (*enclosed in* "[" *and* "]").
Each component is labeled by its offset (in bits) in its closest enclosing aggregate.

In the example in Figure 5, the alias contexts are empty because there is no ambiguity about aliasing. Having alias contexts allows us to summarize function calls with better precision (as opposed to having to make fixed assumptions about aliasing). Now consider the example in Figure 6, which shows how alias contexts can provide better precision. Function `g` returns either `null` or the object that is pointed to by the first parameter, depending on whether `*p1` and `*p2` are aliases.

Checking a call to a trusted function involves a *binding* process and an *update* process. The binding process matches the placeholders with actual abstract locations, and checks whether they meet the obligation. The update process updates the typestates of all actual locations that are represented by the placeholders according to the safety post-conditions.

``PointPtr g(PointPtr *p1, PointPtr* p2){ *p2 = null; return *p1 }```	Safety Pre-condition: `%o0`: <`PointPtr ptr`, {q1}, fo> `%o1`:<`PointPtr ptr`, {q2}, fo> q1: <`PointPtr`, {r1}, fo> Safety Post-condition: [(q1 neq q2), `%o0` : <`PointPtr`, {r1}, ...>] [(q1 eq q2), `%o0` : <`PointPtr`, {`null`}, ...>]

Figure 6: An example of safety pre- and post-conditions with alias contexts.

Our goal is to summarize library functions, which generally do not do very complicated things with pointers. Thus, at present we have focused only on obligations that can be represented as a tree of placeholders. When obligations cannot be represented in this way, we fall back on letting the typestate-propagation phase trace into the body of the function. Tree-shaped placeholders allow the binding process to be carried out with a simple algorithm: The binding algorithm iterates over all formal parameters, and obtains the respective actual parameters from the typestate descriptors at the call site. It then traverses the obligation tree, checks whether the actual parameter meets the obligation, and establishes a mapping between the placeholders and the set of abstract locations they may represent in the store at the callsite.

The binding process distinguishes between may information and must information. Intuitively, a placeholder must represent a location if the binding algorithm can establish that it can only represent a unique concrete location. The algorithm for the updating process interprets each post-condition. It distinguishes a strong update from a weak update depending on whether a placeholder must represent a unique location or may represent multiple locations, and whether the alias context evaluates to true or false. A strong update happens when the placeholder represents a unique location and the alias context evaluates to true. A weak update happens if the placeholder may represent multiple locations or the alias context cannot be determined to be either definitely true or definitely false; in this case, the typestate of the location receives the meet of its typestate before the call and the typestate specified in the post-condition. When the alias context cannot be determined to be either definitely true or definitely false, the update specified by the post-condition may or may not take place. We make the safest assumption via a weak update.

5 Inferring Information about Stack-Allocated Arrays

Determining information about arrays that reside on the stack is difficult because we need to figure out both their types and their bounds. Our previous work [35] required manual annotations of procedures that made use of local arrays. In this section, we describe a method for inferring that a subrange of a stack frame holds an array, and illustrate the method with a simple example.

Figure 7 shows a C program that updates a local array; the second column shows the SPARC machine code that is produced by compiling the program with "`gcc -O`" (version 2.7.2.3). To infer that a local array is present, we examine all live pointers each time the typestate-propagation algorithm reaches the entry of a loop. In the following discussion, the abstract location *SF* denotes the stack frame that is allocated by the `add` instruction at line 2; *SF*[*n*] denotes the point in *SF* at offset *n*; and *SF*[*s*,*t*] denotes the subrange of *SF* that starts at offset *s* and ends at offset *t*-1.

By abstractly interpreting the `add` instructions at lines 3 and 5, we find that `%g3` points to *SF*[*96*] and `%g2` points to *SF*[*176*]. The first time the typestate-checking algorithm visits the loop entry, `%g2` and `%o1` both point to *SF*[*176*] (see the third column of Figure 7). Abstractly interpreting the instructions from line 10 to line 14 reveals that *SF*[*96,100*] stores an integer. The second time the typestate-checking algorithm visits the loop entry, `%g3` points

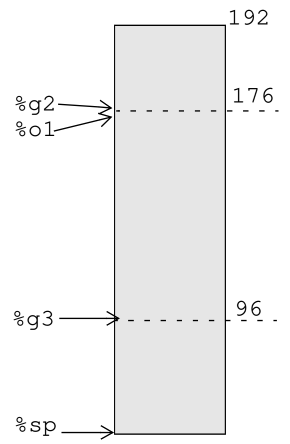
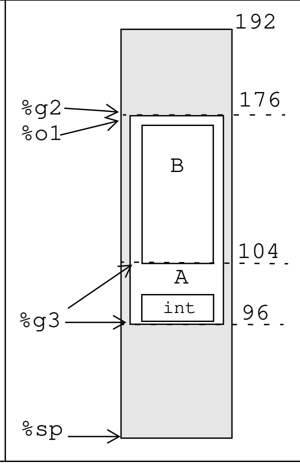
C program	SPARC Machine Language	First Time	Second Time
typedef struct { int f; int g; } s; int main() { s a[10]; s *p = &a[0]; int i=0; while (p<a+10) { (p++)->f = i++; } }	main: add %sp,-192,%sp add %sp,96,%g3 mov 0,%o0 add %sp,176,%g2 cmp %g3,%g2 bgeu .LL3 mov %g2,%o1 .LL4: st %o0,[%g3] add %g3,8,%g3 cmp %g3,%o1 blu .LL4 add %o0,1,%o0 .LL3: retl sub %sp,-192,%sp		

Figure 7: Inferring the Type and Size of a Local Array.
The label .LL4 represents the entry of the while loop.

to either *SF*[*96*] or *SF*[*104*]. We now have a candidate for a local array. The reasoning runs as follows: if we create two fictitious components *A* and *B* of *SF* (as shown in the right-most column in Figure 7), then %g3 can point to either *A* or *B* (where *B* is a component of *A*). However, an instruction can have only one (polymorphic) usage at a particular program point; therefore, a pointer to *A* and a pointer to *B* must have compatible types. The only choice (in our type system) is a pointer into an array. Letting τ denote the type of the array element, we compute a most general type for τ by the following steps:

1. Compute the size of τ. We compute the greatest common divisor (GCD) of the sizes of the slots that are delimited by the pointer under consideration. In this example, there is only one slot: *SF*[96, 104], whose size is 8. Therefore, the size of τ is 8.
2. Compute the possible limits of the array. We assume that the array ends at the location just before the closest live pointer into the stack (other than the pointer under consideration).
3. Compute the type of τ. Assuming that the size of τ we have computed is *n*, we create a fictitious location *e* of size *n*, and give it an initial type $\top$(n). We then slide *e* over the area that we have identified in the second step, *n* bytes at a time—e.g., *SF*[96,176], 8 bytes at a time—and perform a meet operation with whatever is covered by *e*. If an area covered by *e* (or a sub-area of it) does not have a type associated with it, we assume that its type is $\top$. In this example, the τ that we find is

```
struct {
    int m1;
    ⊤(32) m2;
}
```

No more refinement is needed for this example. In general, we may need to make refinements to our findings in later iterations of the typestate-checking algorithm. Each refinement will bring the element type of the array down in the type lattice. In this example, the address under consideration is the value of a register; in general it could be of the form "r_1+r_2" or "r_1 +*n*", where r_1 and r_2 are registers and *n* is an integer.

This method uses some heuristics to compute the possible limits of the array. This does not affect the soundness of this approach for the following two reasons: (i) The typestate-propagation algorithm will make sure that the program is type correct. This will ensure that the element type inferred is correct. (ii) The global-verification phase will verify later that all references to the local array are within the inferred bounds.

Note that it does not matter to the analysis whether the original program was written in terms of an n-dimensional array or in terms of a 1-dimensional array; the analysis treats all arrays as 1-dimensional arrays. This approach works even when the original code was written in terms of an n-dimensional array because the layout scheme that compilers use for an n-dimensional array involves a linear indexing scheme, which is reflected in linear relationships that the analysis infers for the values of registers.

6 Range Analysis

The technique we have used for array bounds checking in our earlier work [35], and techniques such as those described by Cousot and Halbwachs [8] are more precise, but have a higher cost. We describe a simple range analysis that determines safe estimates of the range of values each register can take on at each program point [32].This information can be used for determining whether accesses on arrays are within bounds. We take advantage of the synergy of an efficient range analysis and an expensive but powerful technique that can be applied on demand. We apply the program-verification technique only for the conditions that cannot be proven by the range analysis.

The range-analysis algorithm that we use is a standard worklist-based forward dataflow algorithm. It finds a symbolic range for each register at each program point. In our analysis, a range is denoted by $[l, u]$, where l and u are lower and upper bounds of the form $ax+by+c$ (a, b, and c are integer constants, and x and y are symbolic names that serve as placeholders for either the base address or the length of an array). The reason that we restrict the bounds to the form of $ax+by+c$ is because that array-bounds checks usually involves checking either that the range of an array index is a subrange of $[0, \mathit{length}-1]$, or that the range of a pointer that points into an array is a subrange of $[\mathit{base}, \mathit{base}+\mathit{length}-1]$, where *base* and *length* are the base address and length of the array, respectively. In the analysis, symbolic names such as x and y stand for (unknown) values of quantities like *base* and *length*.

Ranges form a meet semi-lattice with respect to the following meet operation: for ranges $r=[l, u]$, $r'=[l', u']$, the meet of r and r' is defined as $[min(l, l'), max(u, u')]$; the top element is the empty range; the bottom element is the largest range $[-\infty, \infty]$. The function $min(l, l')$ returns the smaller of l and l'. If l and l' are not *comparable* (i.e., we cannot determine the relative order of l and l' because, for instance, $l=ax+by+c$, $l'=a'x'+b'y'+c'$, $x\neq x'$, and $y\neq y'$), *min* returns $-\infty$. The function *max* is defined similarly except that it returns the greater of its two parameters, and ∞ if its two parameters are not comparable.

Operations	$x=x'$, $y=y'$	$x=x'$, $y\neq y'$	$x\neq x'$, $y=y'$	$x\neq x'$, $y\neq y'$
$+_+$	$(a+a')x+(b+b')y+c+c'$	if $(a+a')=0$, $by+b'y'+c+c'$ otherwise, ∞	if $(b+b')=0$, $ax+a'x'+c+c'$ otherwise, ∞	∞
$+_-$		if $(a+a')=0$, $by+b'y'+c+c'$ otherwise, $-\infty$	if $(b+b')=0$, $ax+a'x'+c+c'$ otherwise, $-\infty$	$-\infty$
$-_+$	$(a-a')x+(b-b')y+c-c'$	if $(a-a')=0$, $by-b'y'+c-c'$ otherwise, ∞	if $(b-b')=0$, $ax-a'x'+c-c'$ otherwise, ∞	∞
$-_-$		if $(a-a')=0$, $by-b'y'+c-c'$ otherwise, $-\infty$	if $(b-b')=0$, $ax-a'x'+c-c'$ otherwise, $-\infty$	$-\infty$

Figure 8: Binary Operations over Symbolic Expressions.

We give a dataflow transfer function for each machine instruction, and define dataflow transfer functions to be strict with respect to the top element. We introduce four basic abstract operations, $+$, $-$, $\times$, and $\div$, for describing the dataflow transfer functions. The abstract operations are summarized below, where n is an integer:

$[l, u] + [l', u'] = [l +_- l', u +_+ u']$

$[l, u] - [l', u'] = [l \ -_{-} \ u', u \ \ -_{+} \ l']$

$[l, u] \times n = [l \times n, u \times n]$

$[l, u] \div n = [l \div n, u \div n]$

The arithmetic operations $+_{+}$, $+_{-}$, $-_{+}$, $-_{-}$ over bounds $ax+by+c$ and $a'x' + b'y'+c'$ are given in Figure 8, where a, b, a', and b' are non-zero integers. These arithmetic operations ensure that the bounds are always of the form $ax+by+c$.

Comparison instructions are a major source of bounds information. Because our analysis works on machine code, we need only consider tests of two forms: $w \le v$ and $w = v$ (where w and v are program variables). Figure 9 summarizes the dataflow transfer functions for these two forms. We assume that the ranges of w and v are $[l_w, u_w]$ and $[l_v, u_v]$ before the tests. The function $min_1(l, l')$ and $max_1(l, l')$ are defined as follows:

$$min_1(l,l') = \begin{cases} min(l,l') & \text{if comparable}(l,l') \\ l & \text{otherwise} \end{cases} \quad \text{and} \quad max_1(l,l') = \begin{cases} max(l,l') & \text{if comparable}(l,l') \\ l & \text{otherwise} \end{cases}$$

If a upper bound of a range is smaller than its lower bound, the range is equivalent to the empty range. For the dataflow functions for variables w and v along the false branch of the test $w=v$, we could improve precision slightly by returning the empty range when l_w, u_w, l_v, and u_v are all equal.

Test		w	v
$w = v$	True Branch	$[max_1(l_w, l_v), min_1(u_w, u_v)]$	$[max_1(l_v, l_w), min_1(u_v, u_w)]$
	False Branch	$[l_w, u_w]$	$[l_v, u_v]$
$w \le v$	True Branch	$[l_w, min_1(u_w, u_v)]$	$[max_1(l_v, l_w), u_v]$
	False Branch	$[max_1(l_w, l_v+1), u_w]$	$[l_v, min_1(u_v, u_w-1)]$

Figure 9: Dataflow Functions for Tests.

To ensure the convergence of the range-analysis algorithm in the presence of loops, we perform a widening operation at a node in the loop that dominates the source of the loop backedge. Let r=$[l, u]$ be the range of an arbitrary variable x at the previous iteration and r'=$[l', u']$ be the dataflow value of x at the current iteration. The resulting range will be r''= r ∇ r' where ∇ is the widening operator defined as follows:

$$[l, u] \nabla [l', u'] = [l'', u''], \text{ where } l'' = \begin{cases} -\infty & \text{if } (l' < l) \\ l & \text{otherwise} \end{cases} \quad \text{and} \quad u'' = \begin{cases} \infty & \text{if} (u < u') \\ u & \text{otherwise} \end{cases}$$

We sharpen the basic range analysis with two enhancements. The first enhancement deals with selecting the most suitable spot in a loop to perform widening. The key observation is that for a "do-while" loop (which is the kind that dominates in binary code[2]), it is more effective to perform widening right before the test to exit the loop. In the case of a loop that iterates over an array (e.g., where the loop test is "i < *length*") this strategy minimizes the imprecision of our relatively crude widening operation: the range for i is widened to $[0, +\infty]$ just before the loop test, but is then immediately sharpened by the transfer function for the loop test, so that the range propagated along the loop's backedge is [0, *length*-1]. Consequently, the analysis quiesces after two iterations. The second enhancement is to uti-

2. Although "while" and "for" loops are more common in source code, compilers typically transform them to an "if" with a "do-while" in the "then-part" of the "if". After this transformation has been done, the compiler can exploit the fact that the code in the body of the "do-while" will always be executed at least once if the loop executes. Thus, it is possible to perform code-motion without the fear of ever slowing down the execution of the program. In particular, the compiler can hoist expressions from within the body of the loop to the point in the "then-part" just before the loop, where they are still guarded by the "if".

lize correlations between register values. For example, if the test under consideration is $r < n$ and we can establish that $r = r' + c$ at that program point, where c is a constant, we can incorporate this information into the range analysis by assuming that the branch also tests $r' < n-c$.

		Sum	Paging Policy	Start Timer	Hash	Bubble Sort	Stop Timer	Btree	Btree2	Heap Sort 2	Heap Sort	jPVM	Stack-smashing	jPVM 2	/dev/kerninst	MD5
Number of Each Feature	**Instructions**	13	20	22	25	25	36	41	51	71	95	157	309	315	339	883
	Branches	2	5	1	4	5	3	11	11	9	16	12	89	16	45	11
	Loops (Inner loops)	1	2 (1)	0	1	2 (1)	0	2 (1)	2 (1)	4 (2)	4 (2)	3	7(1)	3	6(4)	5(2)
	Procedure Calls (Trusted Calls)	0	0	1 (1)	1 (1)	0	2 (2)	0	4 (4)	3	0	21 (21)	2	40 (40)	36 (25)	6
	Global Conditions (Bounds Checks)	4 (2)	9	13	14 (2)	19 (10)	17	41 (18)	42 (16)	56 (26)	84 (42)	57 (18)	162 (136)	99 (18)	134 (58)	135 (38)
	Source Language	C	C	C	C	C	C	C	C	C	C	C	C	C	C++	C
Typestate Propagation		0.02	0.05	0.02	0.04	0.04	0.04	0.09	0.11	0.18	0.15	0.63	0.68	3.08	4.60	6.95
Annotation		0.003	0.006	0.006	0.006	0.005	0.007	0.008	0.011	0.015	0.015	0.034	0.033	0.069	0.073	0.083
Range Analysis		0.01	0	0	0.01	0.03	0	0.03	0.03	0.08	0.13	0.13	0.52	0.26	0.79	0.87
Global Verification		0.06	0.16	0.11	0.35	0.18	0.13	0.42	0.39	1.13	2.49	0.78	12.99	1.57	18.6	5.66
TOTAL (Seconds)		0.09	0.21	0.13	0.40	0.25	0.17	0.54	0.54	1.40	2.78	1.58	14.22	4.97	22.92	13.07

Figure 10: Characteristics of the Examples and Performance Results.
The test cases written in C are compiled with `gcc -O` *(version 2.7.2.3),*
and the example /dev/kerninst written in C++ is compiled with Sun Workshop Comipler 5.0.

7 Case Studies

All of the techniques described above, with the exception of the technique from Section 5 to infer sizes of local arrays, have been implemented in our safety-checker for SPARC machine programs [35]. We illustrate the benefits of these improvements on a few example programs. These examples include array sum, start-timer and stop-timer code taken from Paradyn's performance-instrumentation suite [12], 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 [28], bubble sort, two versions of heap sort (one manually inlined version and one interprocedural version), stack-smashing (example 9.b described in [25]), MD5Update of the MD5 Message-Digest Algorithm [22], several functions from jPVM [10] (two cases, where one case includes more functions), and a module in the device driver /dev/kerninst [31] that reads the kernel symbol table.

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, and all array out-of-bounds violations in the /dev/kerninst example. Figure 10 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, perform range analysis, and perform global verification. Figure 10 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), number of global safety conditions (number of bounds checks), and the source language in which each test case is written. Note that the checking of the lower and upper bounds are regarded as two separate safety conditions. The times to verify these examples range from 0.1 seconds to 23 seconds.

The extensions to the typestate system allow us to handle a broader class of real-life examples. Having bit-level representations of integers allow the analysis to deal with instructions that load/store a partial word in the Md5Update and stack-smashing examples. The technique to summarize trusted functions allow the analysis to use summaries of several host and library functions in hash, start- and stop-timer, Btree2, the two jPVM examples, and /dev/kerninst. For these examples, we simply summarize the library functions without checking them. This implies that the examples are safe only if the library functions are safe. In principle, we could check the library code once and use the summaries whenever possible. Subtyping among structures and pointers allows summaries to be given for JNI [9] methods that are polymorphic. For example, the JNI function "`jsize GetArrayLength(JNIEnv* env, jarray array)`" takes the type `jarray` as the second parameter, and it is also applicable to the types `jintArray` and `jobjectArray`, both of which are subtypes of `jarray`. Because all Java objects have to be manipulated via the JNI interface, we model the types `jintArray` and `jobjectArray` as physical subtypes of `jarray` when summarizing the JNI interface functions.

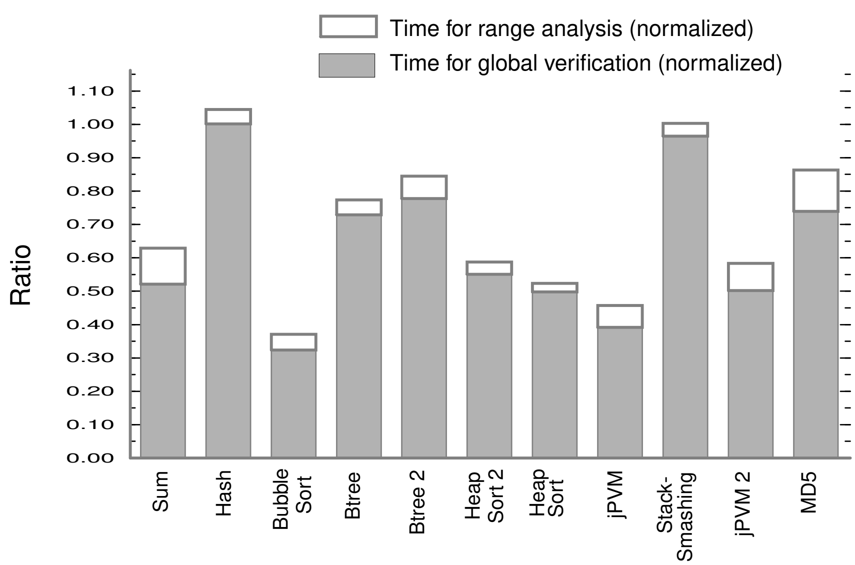


Figure 11: Times to perform global verification with range analysis normalized with respect to times to perform global verification without range analysis.

Symbolic range analysis allows the system to identify the boundaries of an array that is one field of a structure in the MD5 example. When the typestate-propagation algorithm needs information about the range of a register value, we run an intraprocedural version of the range analysis on demand, and the intraprocedural range analysis is run at most once for each function. In the 11 of our test cases that have array accesses, range analysis eliminated 60% of the total attempts to synthesize loop invariants. In 4 of the 11 test cases, it eliminated the need to synthesize loop invariants altogether. The resulting speedup for global verification ranges from -0.4% to 63% (with a median of 37%). Furthermore, in conjunction with improvements that we made to our global-verification phase, range analysis allows us to verify the /dev/kerninst example, which we were not able to handle previously. Figure 11 shows the times for performing global verification, together with the times for performing range analysis (normalized with respect to the times for performing global verification without range analysis). The reason that the analysis of the stack-smashing example is slowed down is because most array accesses in that example are out of bounds. When the array accesses are, in fact, out of bounds, range analysis will not speed up the overall analysis because the analysis still needs to apply the program-verification technique before it can conclude that there are array out-of-bounds violations. Similarly, the reason that hash is slowed down is because only 2 of the 14 conditions are array-bounds checks, and the range analysis cannot prove that the array accesses are within bounds.

8 Related Work

There are several papers that have investigated topics related to the typestate-checking system and symbolic range analysis that we use.

Morrisett *et al* [14,15,17] introduced the notion of typed assembly language (TAL). Our type system and theirs model different language features: For instance, TAL models several language features that we do not address, such as exceptions and existential types. On the other hand, our system models size and alignment constraints, whereas TAL does not. Furthermore, the TAL type system does not support general pointers into the stack, and because stack and heap pointers are distinguished, one cannot declare a function that receives a tuple argument that can accept both a heap-allocated tuple at one call site and a stack-allocated one at another call site [16]. Finally, TALx86 introduces special macros for array subscripting and updating to prevent an optimizer from rescheduling them. (These macros expand into code sequences that perform array-bounds checks). We impose no such restrictions on the idioms that a compiler can employ to implement array subscripting. TAL achieves flow-sensitivity in a different way than our system does; with TAL, different blocks of code are labeled as different functions, and types are assigned to the registers associated with each function. Our system achieves flow-sensitivity by having a different typestate at each instruction.

Mycroft [11] described a technique that reverse engineers C programs from target machine code using type-inference techniques. His type-reconstruction algorithm is based on Milner's algorithm W [13]; it associates type constraints with each instruction in an SSA representation of a program; type reconstruction is via unification. Mycroft's technique infers recursive data-types when there are loops or recursive procedures. We start from annotations about the initial inputs to the untrusted code, whereas his technique requires no annotation. We use abstract interpretation, whereas he uses unification. Note that the technique we use to detect local arrays is based on the same principle as his unification technique. Mycroft's technique currently only recovers types for registers (and not memory locations), whereas our technique can handle both stack- and heap-allocated objects. Moreover, his technique recovers only type information, whereas ours propagates type, state, and access information as well. Our analysis is flow-sensitive, whereas Mycroft's is flow-insensitive, but it recovers a degree of flow sensitivity by using SSA form so that different variables are associated with different live ranges.

Ramalingam *et al* [21] describe an efficient algorithm for decomposing aggregates such as records and arrays into simpler components based on the access patterns specific to a given program. The technique we use to infer local arrays identifies aggregates (arrays) based on the access pattern that is specific to a loop.

Several people have described techniques that can be used to statically check for out-of-bounds array accesses. Cousot and Halbwachs [8] described a method that is based on abstract interpretation using convex hulls of polyhedra. Their technique is precise in that it does not simply try to verify assertions, but instead tries to discover assertions that can be deduced from the semantics of the program. Our range analysis can be regarded as a simple form of Cousot and Halbwachs' analysis with an eye towards efficiency. Our goal is to take advantage of the synergy of an efficient range analysis and an expensive but powerful program-verification technique [30] that can be applied on demand. We apply the program-verification technique only for conditions that cannot be proven by the range analysis.

Verbrugge *et al* [32] described a range-analysis technique called Generalized Constant Propagation (GCP). Our symbolic range analysis differs from GCP in the following respects: GCP uses a domain of intervals of scalars, whereas we use symbolic ranges. GCP attempts to balance convergence and precision of analysis by "stepping up" ranges for variables that have failed to converge after some fixed number of iterations. We perform a widening operation right away for quicker convergence, but sharpen our analysis by selecting suitable spots in loops for performing the widening operation, and also by incorporating correlations among register values. Both GCP and our technique use points-to information discovered in an earlier analysis phase. Our current implementation of range analysis is context-insensitive, whereas GCP is context-sensitive.

Rugina and Rinard [23] also use symbolic bounds analysis. Their analysis gains context sensitivity by representing the symbolic bounds for each variable as functions (polynomials with rational coefficients) of the initial values of formal parameters. Their analysis proceeds as follows: For each basic block, it generates the bounds for each variable at the entry; it then abstractly interprets the statements in the block to compute the bounds for each variable at each program point inside and at the exit of the basic block. Based on these bounds, they build a symbolic constraint system, and solve the constraints by reducing it to a linear program over the coefficient variables from the symbolic bound polynomials. They solve the symbolic constraint system with the goal of minimizing the upper bounds and maximizing the lower bounds.

Bodik *et al* [8] 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. Their goal is to apply their analysis to array bounds checks selectively based on profile information, and fall back on run-time checks for cold code blocks. Wagner *et al* [87] have formulated the buffer-overrun-detection problem as *an integer constraint problem* that can be solved in linear time in practice. Their analysis is flow- and context-insensitive with a goal of finding as many errors as possible.

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] 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).

[3] S. Chandra, and T. Reps. Physical Type Checking for C. *PASTE '99: SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Toulouse, France (September 1999).

[4] 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).

[5] B. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant Context Inference. *ACM Symposium on Principles of Programming Languages.* San Antonio, TX *(*January 1999).

[6] B. Cheng, W. W. Hwu. Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation.*SIGPLAN Conference on Programming Language Design and Implementation.* Vancouver B.C., Canada (June 2000).

[7] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A Certifying Compiler for Java. *SIGPLAN Conference on Programming Language Design and Implementation.* Vancouver B.C., Canada (June 2000).

[8] 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).

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

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

[11] A. Mycroft. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). *8th European Symposium on Programming, ESOP'99*. Amsterdam, The Netherlands (March 1999).

[12] 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).

[13] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* **17**, 3 (1978).

[14] 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).

[15] 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).

[16] 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).

[17] 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).

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

[19] 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).

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

[21] G. Ramalingam, John Field and Frank Tip. Aggregate Structure Identification and Its Application to Program Analysis. *26th ACM Symposium on Principles of Programming Languages*. San Antonio, TX (January 1999).

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

[23] R. Rugina and M. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. *SIGPLAN Conference on Programming Language Design and Implementation.* Vancouver B.C., Canada (June 2000).

[24] 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).

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

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

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

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

[29] *F. Smith, D. Walker, and G. Morrisett.* Alias Types. *European Symposium on Programming*. Berlin, Germany (March 2000).

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

[31] 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).

[32] C. Verbrugge, P. Co, and L. Hendren. Generalized Constant Propagation A Study in C. *6th International Conference on Compiler Construction*. Linköping, Sweden (April 1996).

[33] P. Wadler. A taste of linear logic. Mathematical Foundations of Computer Science, Lecture Notes in Computer Science **711**. Springer-Verlag. Gdansk, Poland (August 1993).

[34] 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).

[35] Z. Xu, B. P. Miller, and T. W. Reps. Safety Checking of Machine Code. *SIGPLAN Conference on Programming Language Design and Implementation.* Vancouver B.C., Canada (June 2000).