SAFETY-CHECKING OF MACHINE CODE

BY

ZHICHEN XU

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

University of Wisconsin—Madison

2001

SAFETY-CHECKING OF MACHINE CODE

Zhichen Xu

Under the supervision of Professor Barton Miller and Professor Thomas Reps

at the University of Wisconsin—Madison

Importing and executing untrusted foreign code has become an everyday occurrence: Web servers download plug-ins and applets; databases load type-specific extensions; and operating systems load customized policies and performance measurement code. Certification of the safety of the untrusted code is crucial in these domains.

I have developed new methods to determine statically whether it is safe for untrusted machine code to be loaded into a trusted host system. My safety-checking technique operates directly on the untrusted machine-code program, requiring only that the initial inputs to the untrusted program be annotated with typestate information and linear constraints. This approach opens up the possibility of being able to certify code produced by any compiler from any source language. It eliminates the dependence of safety on the correctness of the compiler because the final product of the compiler is checked. It leads to the decoupling of the safety policy from the language in which the untrusted code is written, and consequently, makes it possible for safety checking to be performed with respect to an extensible set of safety properties that are specified on the host side.

I have implemented a prototype safety checker for SPARC machine-language

programs, and applied the safety checker to examples (ranging from code that contains just a few branches, to code that contains nested loops, and to code that contains function and method calls). The safety checker was able to mechanically synthesize the loop invariants and check these examples in times ranging from less than a second to dozens of seconds.

Acknowledgments

I would like to thank my advisors Bart Miller and Tom Reps, for their guidance and encouragement. It is a privilege to have both of them as my advisors. I benefit greatly from my daily interactions with them. I thank them for setting a high standard in research, and for their patience with me. I admire their intuitions and their encyclopedic knowledge.

I thank Marvin Solomon and Rastislav Bodik for carefully reading my thesis draft and provide valuable feedback. I thank my entire committee — Bart Miller, Tom Reps, Marvin Solomon, Rastislav Bodik, and Gregory Moses — for a lively and thorough discussion of my dissertation during my defense.

I thank Xiaodong Zhang for being a mentor and a friend, and for encouraging me to apply to better graduate school. I feel deeply in debt to many people. I thank Ari Tamches, Brian Wylie, Phil Roth, Tia Newhall, Vic Zandy, Ling Zheng, Oscar Naim, Karen Karavanic, Andrew Bernat, Alexandre Mirgorodskii, and many others, for being good colleagues and resources.

Finally, I would like to thank my parents Zhenxiang Xu and Meiling Wang for encouraging me to pursue graduate study in the United States. Most of all, I would like to thank my wife Yi Shen and my son Alex Xu. I thank my wife for tolerating my long journey to get a Ph.D. Without her support, I would have never succeeded. I thank my son Alex for giving me much joy.

Contents

List of Figures

Chapter 1

Introduction

Two prevailing trends in software development call for techniques to protect one software component from another. The first trend is *dynamic extensibility*, where a trusted host is extended by importing and executing *untrusted foreign code*. For example, web browsers download plug-ins [65,83]; databases load type-specific extensions for storing and querying unconventional data [40,82]; operating systems load customized policies, general functionality [7,26,55,66,70,75,80,85], and performance-measurement code [85]. Operating systems can download part of an application into the kernel so that the application can perform better. There are even proposals for loading application-specific policies into Internet routers [90]. Certification of the safety of untrusted code is crucial in these domains. The second trend is *component-based software development*, where software components produced by multiple vendors are used to construct a complete application [18] (e.g., COM [51]). The component-based software-development model improves both software reusability and productivity.

However, because the software components can come from different sources, proper protection among software components is essential.

In this thesis, we show how to determine statically whether it is safe for untrusted machine code to be loaded into a trusted *host* system. In contrast to work that enforces safety by restricting the things that can be expressed in a source language (e.g., safe languages, certifying compilers [20,63], and typed assembly languages [58,59,60]), we believe that safe code can be written in any source language and produced by any compiler, as long as nothing "unsafe" is expressed in the machine code. This philosophical difference has several implications. First, it gives the code producer more freedom in choosing an implementation language. Instead of building a certifying compiler for each language, we can certify code produced by a general-purpose off-the-shelf compiler. Second, it leads to the decoupling of the safety policy from the language in which the untrusted code is written. This makes it possible for safety checking to be performed with respect to an extensible set of safety properties that are specified on the host side.

The most important, high-level characteristics of our safety-checking technique are (i) it operates directly on binary code; (ii) it provides the ability to extend the host at a very fine-grained level, in that we allow the untrusted foreign code to manipulate the internal data structures of the host directly; and (iii) it 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, and stack-manipulation violations, in addition to providing the ability for the safety criterion to be extended according

to an *access policy* specified by the host. The host-specified access policy lists the host data that can be accessed and the host functions (methods) that can be called by the untrusted code. This provides a means for the host to grant the "least privilege" that the untrusted code needs to accomplish its task.

Our approach is based on annotating the global data in the trusted host. The type information (more precisely, typestate 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 typestates and linear constraints (i.e., linear equalities and inequalities that are combined with $\wedge$, $\vee$, $\neg$, and the quantifiers $\exists$ and $\forall$). Our analysis uses typestates (as opposed to types) because the condition under which it is safe to perform an operation is a function of not just the types of the operation's operands, but also their states. For example, it is safe to write to a location that stores an uninitialized value, but it is unsafe to read from it. Typestates differ from types by providing information at a finer granularity. Moreover, typestate checking [78,79] differs from traditional type checking in that traditional type checking is a flow-*insensitive* analysis, whereas typestate checking is a flow-*sensitive* analysis. Typestates can be related to *security automata* [4]. In a security automaton, all states are accepting states; the automaton

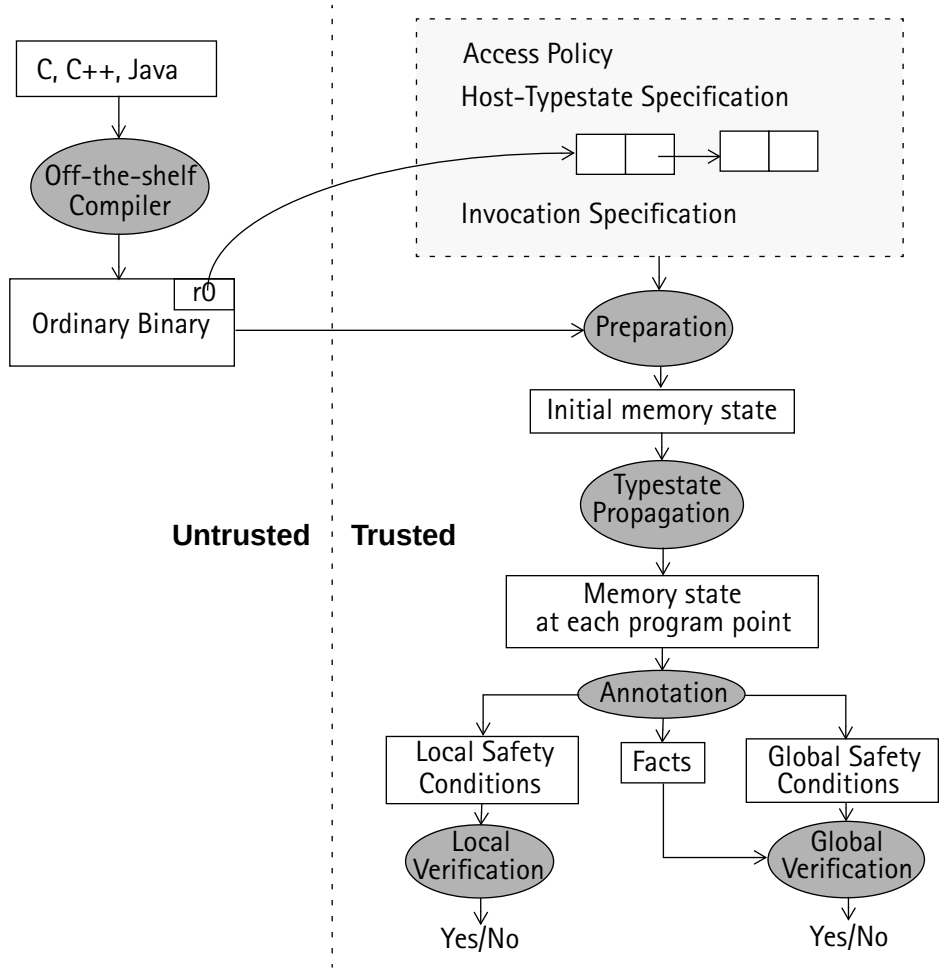


Figure 1.1 The Inputs to and Phases of Our Safety-Checking Analysis.
The dotted vertical line separates the untrusted and trusted worlds.

detects a security-policy violation whenever it reads a symbol for which the automaton's current state has no transition defined. It is possible to design a typestate system that captures the possible states of a security automaton (together with a "security-violation" state). Typestate checking provides a method, therefore, for statically assessing whether a security violation might be possible.

Figure 1.1 illustrates the inputs to and the phases of the safety-checking anal-

ysis. The inputs to the safety-checking analysis include a *host-typestate specification* and an *invocation specification,* in addition to the untrusted code and the host-specified access policy. 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 combination of host-typestate specification, invocation specification and the access policy provides the information about the initial memory state at the time the untrusted code is to be invoked.

The safety-checking analysis consists of five phases: preparation, typestate-propagation, annotation, local verification and global verification. The first two phases find the state(s) on which each instruction operates. The last three phases find the safety conditions each instruction must obey and check the conditions.

The *preparation* phase 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). This phase also produces an interprocedural control-flow graph for the untrusted code. The *typestate-propagation* phase 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 *annotation* phase takes as input the typestate information discovered from the typestate-propagation phase, and

traverses the control-flow graph to annotate 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 *local-verifi cation*phase checks the local safety conditions. The *global-verifi cation* phase verifies the global safety conditions. The global safety conditions perform array bounds checks, null-pointer dereference checks, and address-alignment checks. They are represented as linear constraints. We take advantage of the synergy of an efficient range analysis and an expensive but powerful technique that can be applied on demand for array bounds checks. The range analysis determines safe estimates of the range of values each register can take on at each program point. This information can be used for determining whether accesses on arrays are within bounds. For conditions that cannot be proven just by using the results of the range analysis, we use program-verification techniques. We use the induction-iteration method [84] to synthesize loop invariants if the untrusted code contains loops.

In the above description, the safety-checking analysis both synthesizes and verifies a safety proof. It should be noted that this is just one way to structure the safety-checker. In principle, we could separate the safety-checker into a *proof generator* and a *proof verifi er.* The proof generator generates Proof-Carrying Code (PCC) [64], whereas the proof checker validates the safety proof. In this way, our technique provides a way to lift the current limitations of certifying compilers [20, 63], which produce PCC automatically, but only for programs written in cer-

tain safe source languages.

We have implemented a prototype safety checker for SPARC machine-language programs. We applied the safety checker to several examples (ranging from code that contains just a few branches, to code that contains nested loops, and to code with function and method calls). The safety checker was able to either prove that an example met the necessary safety conditions, or identify the places where the safety conditions were violated, in times ranging from less than a second to tens of seconds on an UltraSPARC machine.

In the remainder of this dissertation, we will call the party that generates the untrusted foreign code the *code producer* (which can be either human or a code-generation tool), and the party who is responsible for the trusted host the *code consumer*.

1.1 Contributions

The major contributions of this thesis are as follows:

1. Our technique opens up the possibility of being able to certify object code produced by off-the-shelf compilers (independent of both the source language and the compiler). We require only that the inputs to the untrusted code be annotated with typestate information and linear constraints.
2. 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.
3. We extend the notion of typestate in several ways: (i) we use typestates to

describe the state information of abstract locations in an abstract storage model; (ii) we extend typestates to include access permissions (which are used to specify the extent to which untrusted code is allowed to access host resources); (iii) in addition to using typestates to distinguish initialized values from uninitialized ones, we also use typestates to track pointers.

4. We propose a typestate-checking system that allows us to perform safety-checking on untrusted machine code that implements inheritance polymorphism via physical subtyping [76]. This work introduces a new method for coping with subtyping in the presence of mutable pointers.
5. We introduce 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 will make the safety-checking technique more scalable.
6. We present a technique to infer information about the sizes and types of stack-allocated arrays.
7. We describe a symbolic range analysis that is suitable for propagating information about array bounds. Range analysis can speed up safety checking because it is generally much less expensive than the program-verification techniques that we use to bounds checks.
8. We describe a prototype implementation of the safety-checking technique, and experimental studies to evaluate it.

The work described in thesis focuses on enforcing fine-grained memory protec-

tion, which allows us 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 [24,39]. Safety checking is less ambitious than verification of correctness.

1.2 Organization of Dissertation

The dissertation is organized into eight chapters. We begin with a discussion of related work in Chapter 2. In Chapter 3, we describe the safety properties we enforce, and the notion of an access policy. We present an overview of our safety checking analysis by means of a simple example.

Chapters 4 to 6 describe the five phases of our safety-checking analysis starting from the second phase. In Chapter 4, we describe the second phase of the safety-checking analysis. We describe an abstract storage model used in the analysis (in particular, a typestate system). We present the typestate-checking analysis that recovers typestate information at each program point in the untrusted code. The typestate-checking system allows us to check the safety of untrusted machine code that implements inheritance polymorphism via physical subtyping. Moreover, we describe several techniques that make the safety-checking analysis more precise and efficient.

In Chapter 5, we present the details of the annotation and local-verification phases of our analysis. In Chapter 6, we describe the global-verification phase. We present the induction-iteration method for synthesizing loop invariants and

our enhancements to it, and also describe a symbolic range analysis for array bounds checking.

In Chapter 7, we present our experience with the safety-checking technique gained from using a prototype implementation for untrusted code written in SPARC machine languages on a few case studies.

Finally, in Chapter 8, we present our conclusions and suggest some directions for future research.

Chapter 2

Related Work

In this chapter we discuss the research efforts that are most closely related to this dissertation. In Section 2.1, we survey techniques to enforce safe program execution. We examine techniques that can be used to statically check for array out-of-bounds violations in Section 2.2. We discuss techniques for synthesizing loop invariants in Section 2.3.

2.1 Safety Checking

Techniques to enforce code safety fall into three categories: dynamic, static, and hybrid. Static techniques are potentially more efficient at run-time than dynamic and hybrid techniques because static techniques incur no run-time overhead, but they can be less precise because they will have to reject code that cannot be determined to be safe statically. A dynamic technique incurs run-time cost and may require corrective actions in the case that a safety violation is detected at run-time. Hybrid techniques tend to be more efficient than dynamic techniques, and like dynamic techniques, they may require corrective actions in the

presence of a safety violation. We survey the related work and compare our work with static techniques that are closest to ours.

2.1.1 Dynamic Techniques

Two safety issues must be addressed when a dynamic technique is used to enforce code safety. First, safety violations must be detected. Second, corrective action must be taken after a violation is detected at run-time. This corrective action can be as simple as terminating the offending code, or can be much more complex if the offending code accesses shared data structures in the trusted host.

One extreme of dynamic techniques is safety through interpretation, where a virtual machine (VM) interprets the untrusted code and checks the safety of each instruction at run-time. The BSD network packet filter utility [49, 55], commonly referred to as BPF, is such an example. It defines a language that is interpreted by the operating system's network driver. The interpreter checks, at run-time, that all references to memory are within the bounds of the packet data or a statically allocated scratch memory. Interpretation incurs high run-time cost. For example, BPF is about 10 times slower than versions written in the statically checked proof-carrying code [64]. Moreover, languages that are designed to be interpreted are usually small, and with limited control and data structures, makes them unsuitable for general-purpose use.

A simple way to enforce the safety of untrusted code is to isolate it in a hardware-enforced address space, similar to the way operating system kernels protect themselves from user-level applications [8]. In this approach, the hardware and

the operating system kernel prevent code in one address space from corrupting code and data in another address space. To prevent the untrusted code from leaving the host software in an inconsistent state (e.g., terminating without releasing resources it has acquired), the untrusted code must interact with the host software through a restricted interface. Apart from the apparent limitation of requiring special hardware support, a major disadvantage of this approach is its high run-time cost. A cross-address-space call requires (at least) a trap into the operating system kernel, copying of arguments from the caller to the callee, saving and restoring registers, switching hardware address spaces, and dispatch back to user level [93].

Software Fault Isolation (SFI) [93] uses pure software techniques to achieve much of the same functionality as hardware-enforced address spaces, but at a much lower cost. A form of SFI, *sandboxing,* ensures that the high bits of a memory address match those of the sandbox region assigned to the foreign code. VINO [75] and Exokernel [26] are two systems that use sandboxing to ensure that extensions downloaded into the OS are safe. Both SFI and hardware-enforced address spaces provide protection by isolating the untrusted code in its own protection domain, and restricting the interface through which it can interact with the host software. Since SFI modifies the binary code directly, it is independent of the source language. Like hardware-enforced address spaces, the protection provided by SFI is coarse-grained, and is not appropriate for a system with fine-grained sharing. SFI incurs low run-time overhead on processors with a large number of registers, because less register spilling is needed to free up registers

for sandboxing. However, if the untrusted code interacts frequently with code in the host environment (or other untrusted components residing in different protection domains) and the read operations must be checked also, the overhead of run-time checking can amount to 20% as opposed to only a few percent when only write operations are checked [93]. Checking read operations is necessary because reads to certain memory-mapped devices could be destructive. Finally, SFI can be difficult to implement correctly; for example, it is hard to prevent code from modifying itself, and to protect the contents of the stack.

Leroy and Rouaix [46] have proposed a theoretical model for systematically placing type-based run-time checks into interface routines of the host code to provide fine-grained memory protection. Their technique checks the host and requires that the source of the host API be available. Safety requirements are specified by enumerating a set of predetermined sensitive locations and invariants on these locations.

Our technique is related to Leroy and Rouaix's technique in that our technique is also type-based. However, we rely on static analysis rather than run-time checking. In addition, our technique works on untrusted binary code whereas their technique instruments the host API at the source level. Our model of a safety policy (see Chapter 4) is more general than theirs. Finally, they perform type checking whereas we perform typestate checking (see Chapter 4). Typestates provide finer grained information than types, and typestate checking is a flow-*sensitive* analysis, whereas traditional type checking is a flow-*insensitive* analysis. Hence, our technique is potentially more precise.

2.1.2 Static Techniques

Static techniques to ensure code safety have two advantages. First, the code is potentially more efficient because no run-time checks are involved. Second, no corrective action is needed since the code can never misbehave. Static techniques to enforce code safety range from those that provide accountability, to techniques that use formal methods to verify that a binary conforms to its specification in logic (correctness checking), to techniques that verify that a binary conforms to certain safety properties (e.g., type safety), to techniques that emphasize finding potential bugs rather than enforcing full safety.

Techniques that check the correctness of a program are hard to automate. Techniques that check that a program has specific safety properties are more manageable than correctness checking, but can still be very expensive. Techniques that focus on finding potential bugs can, for better analysis efficiency, rely on analyses that are neither sound nor complete.

The simplest static technique for enforcing safety is through personal authority. For example, Microsoft's ActiveX [2] uses digital signatures to record information about the origin of the code. SVR4 kernels allow users with super-user privilege to install kernel modules (such as device drivers) into the kernel. This approach provides accountability rather than safety.

Several research projects have used formal methods to verify that binary code conforms to its specification (as a logical formula), and that it has certain memory safety properties [11, 19]. Clutterbuck and Carre [19] describe a technique to prove that programs written in a subset of Intel 8080 machine language conform

to their specifications. Their technique uses the SPADE software tools, which work on programs defined in SPADE's FDL, the SPADE program modelling language. Their safety-checking analysis uses flow-analysis and program-verification techniques. The flow analyses check for such problems as unreachable code, code from which the exit cannot be reached, multiple-entry loops, use of undefined variables, unused definitions, and redundant tests. SPADE's program verifier checks that a program conforms to its specification in logical formulae.

Boyer and Yu [11] have described a different approach to prove that a machine-code program is memory safe and consistent with its mathematical specification. Their technique models the semantics of a subset of MC68020 instruction set architecture (ISA) in meticulous detail by giving the machine code an explicit formal operational semantics. This operational semantics is given in the logic of their automated reasoning system. Analogous to supplying loop invariants with Floyd-style verification conditions, their approach requires manual construction of lemmas.

A major limitation of using general theorem-proving techniques is that proving the validity of an arbitrary predicate (in first-order logic) is undecidable. In addition, proving that a program containing loops satisfies a given pre- and post-condition using Floyd-style verification conditions involves synthesizing loop invariants, which, in general, cannot be done mechanically.

Instead of proving that a program conforms to its specification (i.e., is correct), which is hard to accomplish mechanically in general, several projects focus on verifying that a piece of untrusted code has specific safety properties. Examples

of these projects include Proof-Carrying Code (PCC) [64], the Certifying Compiler [20, 63], Typed-Assembly Language (TAL) [58,59,60], and our work.

PCC is based on the observation that it is generally faster and easier to validate a proof than to generate one. With PCC, a code producer provides code, along with a proof that the code has certain safety properties. Necula and Lee have used PCC to statically check the safety of network packet filters, and to provide safe native extensions to ML. A major advantage of PCC is that safety only depends on the correctness of a proof checker that is relatively small, and no trusted third party is needed. PCC is "tamper proof" in that any change that either makes the code unsafe or the proof invalid will be identified by the proof checker. PCC also has the ability to associate proofs to the end-product, i.e., the machine code. However, manual generation of proofs can be tedious and error-prone. For each type of safety property considered, a proof system is needed. Furthermore, adding proofs to the code can considerably increase the size of the code (3 to 7 times the original size).

To avoid manual construction of PCC, Necula and Lee [63] introduce the notion of a certifying compiler, which compiles a high-level programming language program into PCC. Their prototype compiler, TouchStone, compiles a safe subset of C into assembly code that carries proofs about type safety. They show that loop invariants for type safety (without considering array bounds checks) can be generated automatically by the compiler. Their work shows that the relevant operational content of simple type systems may be encoded using extensions to first-order predicate logic.

Instead of relying on a logic system to encode types, Morrisett *et al* [58,59,60] introduced the notion of typed assembly language (TAL). In their approach, type information from a high-level program is incorporated into the representation of the program in a platform-independent typed intermediate form, and carried through a series of transformations down to the level of the target code. The compiler can use the type information to perform sophisticated optimizations. Certain internal errors of a compiler can be detected by invoking a type-checker after each code transformation. A compiler that uses typed assembly language certifies type safety by ensuring that a well-typed source program always maps to a well-typed assembly program.

Checking full safety can be time consuming because the analyses, at a minimum, have to be sound. A static debugger uses static analysis to find unsafe operations rather than to guarantee full safety. It is willing to make use of analyses that are neither sound nor complete in the interest of efficiency.

Flanagan *et al* [30] describe an interactive static debugger for Scheme that can be used to identify program operations that may cause run-time errors, such as dereferencing a null pointer, or calling a non-function. The program analysis computes value-set descriptions for each term in the program, and constructs a value flow graph connecting the set descriptions. Evans [27] describes extensions to the LCLint checking tool (a tool for statically checking C programs, and a tool that can perform stronger checking than any standard version of lint) [28] to detect dynamic memory errors, such as dereferencing null pointers, failure to allocate or deallocate memory, uses of undefined or deallocated storage, and dan-

gerous or unexpected aliasing. His technique uses interface annotations to avoid expensive interprocedural analysis and to reduce the amount of error messages. In his analysis, loops are treated as though they were conditional statements.

Detlefs *et al* [23] 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). Their analysis makes use of linear constraints, automatically synthesizes loop-invariants to perform bounds checking, and is parameterized by a policy specification. 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.

2.1.3 Our Work vs. Related Static Techniques

The static techniques that are closest to ours are the certifying compiler and typed-assembly language.

The most prominent difference between our approach and the certifying compiler (or the TAL) approach is a philosophical one. The certifying compiler approach enforces safety by preventing “bad” things from being expressible in a source language. For example, both the safe subset of C of the Touchstone compiler and the Popcorn language for TALx86 [60] do not allow pointer arithmetic, pointer casting, or explicit deallocation of memory. In contrast, we believe that

safe code can be written in any language and produced by any compiler, as long as nothing "bad" is said in the code.

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

Our approach does introduce an additional variable into the process that is only partially within the programmer's control — namely, the code-generation idioms that a particular compiler uses could be ones that defeat the techniques used in our system. This is because the implementation of our analyses may rely on (or not aware of) some idioms the compiler uses.

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

In addition to these high-level differences, there are a few technical differences. Our safety checker can be viewed as a certifier that generates proofs by first recovering type information that (may have) existed in the source-language program (an embodiment of a suggestion made by Necula and Lee [63, p. 342]). The approach used in our safety checker differs from that used in the Touchstone

compiler in the following respects: First, Touchstone replaces the standard method for generating verification conditions (VCs), in which formulae 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. (See Chapter 4 for a description of typestate checking, and Chapter 6 for a description 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 formulae are pushed in the forward direction to generate strongest post-conditions; in a forward VC-generation phase, quantifiers accumulate—forcing one to work with larger and larger formulae. Second, our safety-checking analysis mechanically synthesizes loop invariants for bounds checking and alignment checking, whereas Touchstone generates code that contains explicit bounds checks and then removes those checks that it can prove to be redundant.

Comparing with TAL, our type system and that of TAL 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, which TAL does not. Furthermore, the TAL type system does not support general pointers into the stack, and because stack and heap pointers are distinguished by TAL, 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 [59]. 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. Despite the differences, it is interesting to note that if our safety checker were to be given programs written in typed assembly language rather than in an untyped machine language, less work would be required to recover type information and to perform overload resolution (although we would still have to propagate state and access information). This also applies to Java bytecode [47], where type information is contained in the bytecode instructions themselves.

Finally, neither Touchstone nor the Popcorn compiler of TALx86 track aliasing information. We have introduced an abstract storage model and extended typestate checking to also track pointers. As a result, the analysis we provide is more precise than that used in Popcorn and Touchstone.

Our work is also related to the work of Detlefs *et al* [23] in that both their technique and ours make use of linear constraints, automatically synthesize loop-invariants to perform bounds checking, and are parameterized by policy specifications. However, their safety-checking analysis works on source-language pro-

grams and makes use of analyses that are neither sound nor complete. Their policy specifications are less general than our access policies, which are given in terms of regions, categories, and access permissions (see Chapter 4).

2.1.4 Hybrid Techniques

A technique that combines both static and dynamic checking requires fewer run-time checks than dynamic techniques, but still needs corrective actions because faults can still occur at run-time. Examples that use hybrid techniques for safety checking include safe languages, such as Java [34], Mesa [50], and Modula 3 [37].

A safe language has well defined semantics so that all valid programs written in the language are guaranteed to have certain safety properties. It employs both static and run-time measures to avoid operations that are potentially harmful. Systems that use safe languages for system extensions include Pilot [70], which runs programs written in Mesa, HotJava Web Browser, which can be extended with applets written in Java, and the SPIN extensible OS [7], which can be extended with modules written in Modula 3.

In a safe language, the safety property is build into the language. A safe language usually relies on strong typing to enforce fine-grained memory protection and data abstraction. Techniques based on types and programming-language semantics (including PCC, certifying compilers, and safe languages), offer much finer-grained access control and flexibility than those based on physical means,

such as SFI and hardware-enforced address spaces. Types correspond more naturally to the computer resources that we want to protect.

Safe languages prohibit certain "bad things" from happening by restricting the expressiveness of the language, which, as a consequence, also restricts the applicability of the technique to such languages as C or assembly code. Moreover, even for general-purpose type-safe languages, such as Java, there are occasions when some functionality has to be implemented in "low-level" languages such as C or assembly code [41].

2.2 Array Bounds Checking

A basic requirement of safe program execution is that all array accesses should be within their bounds. Array bounds checks are also essential for enforcing program security. For example, buffer-overrun vulnerabilities, which allow a malicious user to overrun the stack contents to circumvent the computer's security checks, have long plagued security architects. Wagner *et al* [91] report that buffer overruns account for up to 50% of today's vulnerabilities (based on data from CERT advisories over the last decade).

Techniques for performing array bounds checks include techniques for optimizing array bounds checks [35,45,48], symbolic analyses that compute the bounds of index for array references [10,21,38,72,86,91], and program-verification techniques [84]. In the next few sections, we describe work in each of these three areas.

2.2.1 Techniques for Optimizing Array Bounds Checks

Markstein *et al* [48] have developed an analysis technique that first moves array bounds checks out of a loop, and then eliminates the checks if the analysis can determine that there are no array out-of-bounds violations. Their analysis places the array bounds checks outside of the loop, modifies the loop-control condition so that it guarantees that no array out-of-bounds violations will occur, and places a test at the loop exit to ensure that the loop will perform the same number of iterations as in the original program.

Gupta [35] also described a technique for optimizing array bound checks. Gupta's optimizations reduce the program execution time and the object code size through elimination of redundant checks, propagation of checks out of loops, and combination of multiple checks into a single check. His analysis is performed on a reduced control-flow graph that consists of only the minimal amount of data flow information for range-check optimizations.

Kolte and Wolfe [45] present a compiler-optimization algorithm to reduce the run-time overhead of array bounds checks. Their algorithm is based on partial redundancy elimination and incorporates previously developed algorithms (including those that were described by Gupta [35]) for array bounds checking optimizations.

The above techniques can be used to perform array bounds checking by first introducing code to perform the checks, and then checking whether such code can be optimized away.

2.2.2 Techniques for Propagating Information about Array Bounds

The algorithms that rely either on dataflow analysis or on abstract interpretation to propagate information about array bounds vary both in sophistication of the assertions and the rules used for propagating and combining the assertions. For example, the assertions used by Verbrugge *et al* [86] are intervals of scalars, whereas Cousot and Halbwachs [21] use convex polyhedra, which can track correlations between variables. Wagner *et al* [91] use flow-insensitive analysis for better analysis efficiency, whereas Verbrugge *et al* [86] use flow- and context-sensitive analysis.

Harrison [38] uses compile-time analysis to reduce the overhead due to range checks. Compile-time techniques for range propagation and range analysis are employed yielding bounds on the ranges of variables at various points in a program. Harrison's technique propagates both ranges that are scalar intervals, and ranges with simple symbolic bounds. The range information is used to eliminate redundant range checks on array subscripts.

Verbrugge *et al* [86] described a range-analysis technique called Generalized Constant Propagation (GCP). GCP uses a scalar interval domain. It employs a flow- and context-sensitive analysis. It attempts to balance convergence and precision of the analysis by " steppingup" ranges (decreasing the lower bound or increasing the upper bound) for variables that have failed to converge after some fixed number of iterations. GCP uses points-to information discovered in an earlier analysis phase.

Rugina and Rinard [72] 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 *at al* [10] 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* [91] 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. Cousot and Halbwachs [21] 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.

We also propose a range analysis for array bounds checking (see Section 6.7). Our range analysis is closest to GCP, but differs from GCP in the following respects: We use a domain of symbolic ranges. 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.

2.2.3 Program-Verification Techniques

Suzuki and Ishihata [84] and German [32] used Floyd-style program verification techniques to verify the absence of array out-of-bound violations in programs. Floyd-style program verification relies on the system's ability to synthesize loop invariants automatically. Suzuki and Ishihata introduced a method called *induction iteration* for synthesizing loop invariants, whereas German's method relies some simple heuristics. Both Suzuki and Ishihata, and German's methods were developed for structured source-level programs.

2.3 Synthesis of Loop Invariants

A major problem in building an automatic verifier that does not require any programmer-supplied annotations is that the system must synthesize loop invari-

ants. There are several ways to synthesize loop invariants automatically: using heuristics, difference equations, abstract interpretation, running the program in a test-oriented fashion [13], and the induction-iteration method [84].

Katz and Manna [43] and Wegbreit [92] both describe the use of heuristics to synthesize loop invariants. In this approach, back-substitutions are performed, starting with the postcondition, to produce trial loop predicates. Trial loop predicates that are not loop invariants are modified according to various heuristics to generate better trial predicates. Many of the heuristics are domain specific. They have shown examples in the domain of integers and integer arrays.

Synthesizing loop invariants using difference equations [25] proceeds in two steps: (i) finding an explicit expression for each variable after t iterations of the loop, and (ii) eliminating t to obtain invariants.

The abstract interpretation method [92] works forward from the precondition. Since the precondition is known to hold, it can be treated as data and submitted as input to an appropriate evaluator. In this evaluator, all operators are treated as operations on predicates in some abstract interpretation, taking predicates as arguments and delivering a predicate as their result. When the evaluator encounters a conditional, it either chooses one of the alternatives (if the current state logically implies either the decision predicate or its negation), or otherwise control splits into parallel branches. Junction nodes are handled by a sequence of two operations: first the predicates on the input arcs are merged; then the result of this merge is joined with the previous predicate on the output edge to form a new predicate on the output edge.

Loop invariants can also be synthesized by running the program in a test-oriented fashion [89], which consists of three steps: the first step selects several values of input variables; the second step runs the program for each of these inputs and collects the values of the output variables at each program point; the third step tries to establish relations among the variables.

The induction-iteration method of Suzuki and Ishihata uses *weakest liberal preconditions* for synthesizing loop-invariant. For each postcondition of a loop that needs to be verified, it inductively synthesizes a loop invariant that (i) is true on entry to the loop and (ii) implies the postcondition.

We have extended the induction-iteration method for machine-language programs, for nested loops, and for interprocedural verification. A description of the induction-iteration method and our extensions to it can be found in Chapter 6.

We adopt the induction-iteration method because it is mechanical, and because the assertions we need to prove are less general than those required to prove that a program conforms to its specification.

Chapter 3

Overview

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. The chief advantage of this approach 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 and C++. 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.

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 [64]; 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 pro-

ducer (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 [20,63], which produce PCC automatically, but only for programs written in certain safe source languages.

To mechanically synthesize and verify a safety proof for a piece of untrusted code, our analysis starts with a description of the initial inputs to the untrusted code and an access policy. It abstractly interprets the untrusted code to produce a safe approximation of the memory state at each program point. These memory states are described in an abstract storage model. Given the information discovered at each program point, our analysis annotates each instruction with the safety conditions the instruction must obey, and then verifies these conditions.

In the reminder of this chapter, we describe the safety properties we enforce and the notion of an access policy. We describe an abstract storage model, and the inputs to our safety-checking analysis. We present an overview of our safety-checking analysis by means of a simple example.

3.1 Safety Properties and Policy

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-specifi ed access policies*

The default safety conditions enforce fine-grained memory protection and data abstraction based on strong typing. The default safety conditions check for type

violations, array out-of-bounds violations, address-alignment violations, uses of uninitialized values, null-pointer dereferences, and stack-manipulation violations. They ensure that the untrusted code will not forge pointers, and that all operations in the untrusted code will operate on only values of proper types and with a proper level of initialization.

An access policy provides additional flexibility by allowing the host to specify the host data that can be accessed and the host functions (methods) that can be called by the untrusted code. It provides a means for the host to specify the " least privilege" that the untrusted code needs to accomplish its task. This can minimize the potential damages the untrusted code may do to the trusted host.

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

An access policy is specified by a classification of the memory locations into regions, and a list of triples of the form [Region : Category : Access Permitted]. A Region can be as large as an entire address space or as small as a single variable. The Category field is a set of types or aggregate fields. The Access field can be any subset of *r*, *w*, *f, x*, and *o,* meaning readable, writable, followable, executable, and operable, respectively.

In our model, *r* and *w* are properties of a location, whereas *f*, *x*, and *o* are prop-

erties of the value stored in a location. The access permission *f* is introduced for pointer-typed values to indicate whether the pointer can be dereferenced. The access permission *x* applies to values of type "pointer to function" (i.e., values that hold the address of a function) to indicate whether the function pointed to can be called by the untrusted code. The access permission *o* includes the rights to "examine", "copy", and perform other operations not covered by *x* and *f*.

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

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

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

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

[*H* : `thread.next` : *rfo*]

The above model can be used to specify a variety of different safety policies. For example, we can specify something roughly equivalent to *sandboxing* [93]. The original sandboxing model partitions the address space into protection domains, and modifies a piece of untrusted code so that it accesses only its own

domain. In our model, sandboxing boils down to allowing untrusted code to access memory only via valid addresses in the untrusted data region, but otherwise to examine, and operate on data items of any type. Because an address of a location in the host region cannot be dereferenced, side-effects are confined to the untrusted region. Our approach differs from sandboxing in that it is purely static, and it does not make any changes to the untrusted code.

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

The safety properties and policies are introduced to ensure that the integrity of the host environment will not be violated and that host resources will not be accessed improperly. A safety policy can also include safety postconditions for ensuring that certain invariants defined on the host data are restored by the time control is returned to the host.

3.2 An Abstract Storage Model

We introduce an *abstract storage model* for describing the memory states at each program point. The abstract storage model provides the abstract domain for our safety-checking analysis. This model includes the notion of an *abstract store* and *linear constraints*.

An abstract store is a total map from *abstract locations* to *typestates*. An abstract location summarizes one or more physical locations (e.g., heap- and stack- allocated objects) so that the analysis has a finite domain to work over. An abstract location has a name, a size, an alignment, and optional attributes *r* and *w* to indicate if the abstract location is readable and writable according to the access policy. A typestate describes the type, state, and access permissions of the value stored in an abstract location. The typestates form a meet semi-lattice. (Typestates are described in Section 4.1.)

The linear constraints are linear equalities and linear inequalities combined with logical operators and quantifiers. They are used to represent safety requirements such as array bounds checks, address alignment checks, and null-pointer checks.

3.3 The Inputs to the Safety-Checking Analysis

The inputs to our safety-checking analysis include the untrusted code, the host-specified access policy (described in Section 3.1), a *host-typestate specification,* and an *invocation specification.* All inputs except for the untrusted code are provided by the host. A host-typestate specification provides information regard-

ing how functions (methods) in the host can be called. Together with the access permission x given in the access policy, they specify what host functions can be called and how they can be called.

A host-typestate specification includes a data aspect and a control aspect. The data aspect describes the type and the state of host data before the invocation of the untrusted code. The control aspect provides safety preconditions and postconditions for calling host functions and methods.

The safety preconditions and postconditions are given in the form of *placeholder* abstract locations. The typestate and size of a placeholder abstract location in a safety precondition represent obligations that the corresponding actual parameter must provide. The placeholder abstract locations in the postconditions specify the typestates of the corresponding locations after the execution of the function. Verifying the safety of a call into a host function (method) involves a binding process that matches the actual parameters with the placeholder abstract locations in the safety preconditions, and an update process that computes the memory state after the invocation of the call based on the safety postconditions. A detailed description of how to summarize calls to trusted functions is given in Section 4.4.

An invocation specification provides the binding information between the resources in the host and the registers and memory locations that represent the parameters of the untrusted code. The host-typestate specification, the invocation specification, and the access policy, together, provide information about the initial memory state at the entry of the untrusted code.

3.4 The Phases of the Safety-Checking Analysis

Starting from the initial memory state, our analysis will abstractly interpret the untrusted code to find a safe approximation of memory state at each program point. The approximations of memory states are described using the abstract storage model. Once our analysis finds the memory state at each program point (i.e., a description of the state(s) on which each instruction operates), we use the default safety conditions and the access policy to attach a safety predicate to each instruction, and check whether each instruction obeys the corresponding safety predicate.

UNTRUSTED CODE	
1: mov %o0,%o2	// %o2=%o0
2: clr %o0	// %o0= 0
3: cmp %o0,%o1	//
4: bge 12	// if (%o0 ≥ %o1) goto 12
5: clr %g3	// %g3= 0
6: sll %g3, 2,%g2	// %g2= 4 *x* %g3
7: ld [%o2+%g2],%g2	// %g2= [%o2+%g2]
8: inc %g3	// %g3= %g3 + 1
9: cmp %g3,%o1	//
10:bl 6	// if (%g3 < %o1) goto 6
11:add %o0,%g2,%o0	// %o0 = %o0 + %g2
12:retl	
13:nop	

Figure 3.1 A Simple Example: Summing the Elements of an Integer Array.

The safety-checking analysis consists of five phases: preparation, typestate-propagation, annotation, local verification, and global verification. We illustrate these phases informally by means of a simple example. Figure 3.1 shows a piece of untrusted code (in SPARC assembly language) that sums the elements of an

integer array. This example will be used a running example throughout the entire thesis.

Figure 3.2 shows the host-typestate specification, the access policy, and the invocation specification. In Figure 3.2, the host-typestate specification states that ap is the base address of an integer array of size n, where $n \geq 1$. We have used a single abstract location e to summarize all elements of the array ap. The safety policy states that ap and e are in the *V* region, that all integers in the *V* region are readable and operable, and that all base addresses to an integer array of size n in the *V* region are readable, operable, and followable. The invocation specification states that ap and the size of ap will be passed through the registers %o0 and %o1, respectively. The code uses three additional registers, %o2, %g2, and %g3.

HOST TYPESTATE	ACCESS POLICY	INVOCATION
e: <int, initialized, ro> ap: <int [*n*], {e}, rfo> {*n*≥1}	V = {e, ap} [V : int : ro] [V : int [*n*] : rfo]	%o0 ← ap %o1 ← *n*
ap *is an integer array of size n, where n*≥1. e *is an abstract location that summarizes all elements of* ap.	ap *and* e *are in the V region. All integers in the V region are readable and operable. All base addresses to an integer array of size* n *in the V region are readable, operable, and followable.*	ap *and the size of* ap *will be passed through the registers* %o0*, and* %o1*, respectively.*

Figure 3.2 Host-typestate specification, invocation specification, and access policy.

(Note that given the annotation that n is a positive integer before the invocation of the untrusted code, the test at lines 3 and 4 in Figure 3.1 is redundant. However, our technique is based on annotating the initial inputs to the untrusted

code, and it makes no assumption about how much optimization has been done to the untrusted code.)

3.4.1 Preparation

The *preparation* phase takes the host-typestate specification, the access policy, and the invocation specification, and translates them into *initial annotations* that consist of linear constraints and the typestates of the inputs. The initial annotation gives the initial abstract store at the entry of the untrusted code. The preparation phase also constructs an interprocedural control graph for the untrusted code.

INITIAL TYPESTATE	INITIAL CONSTRAINTS
e:<int, initialized, ro> %o0:<int [*n*], {e}, rwfo> %o1:<int, initialized, rwo>	$n \geq 1 \wedge n$=%o1

Figure 3.3 Initial Annotations.
For convenience, we have listed the access permissions *r* and *w* of an abstract location together with the *f, x, o* permissions of the value that is stored in the location.

For the example in Figure 3.1, the initial annotations are shown in Figure 3.3. The fact that the address of ap is passed via register %o0 is described in the second line in column 1, where the register %o0 stores the base address of the integer array and points to e. The fact that the size of ap is passed via the register %o1 is captured by the linear constraint "*n*=%o1".

Note that %o0 and %o1 both have the *r* and *w* access permissions. These refer to the registers themselves (i.e., the untrusted code is permitted to read and change the value of both registers). However, array ap cannot be overwritten

because the access permission for e, which acts as a surrogate for all elements of ap, does not have the *w* permission.

Also note that in machine code, a register can be used to store values of different types at different program points. In our model, a register or a memory location on the untrusted code's stack is always writable.

3.4.2 Typestate Propagation

The *typestate-propagation* phase takes the interprocedural control flow graph of the untrusted code and the initial annotations as inputs. It abstractly interprets the untrusted code to annotate each instruction with an abstract representation of the memory contents using the abstract storage model. The abstract representation of memory characterizes the memory state before the execution of that instruction.

For our example, this phase discovers that the instruction at line 7 is an array access, with %o2 holding the base address of the array and %g2 representing the index. The instruction at line 7 loads an integer from e and stores it in the register %g2. Figure 3.4 summarizes the memory state (the typestates of the abstract locations) before the execution of the load instruction at line 7. In Chapter 4, we will elaborate the typestate-checking system for the Phase 2 of our analysis.

3.4.3 Annotation

The *annotation phase* takes as input the typestate information discovered in Phase 2, and traverses the untrusted code to annotate each instruction with safety preconditions and with assertions. The safety preconditions are divided

ABSTRACT STORE
e:<int, initialized, ro>
%o1:<int, initialized, rwo>
%o2:<int [*n*], {e}, rwfo>
%g2:<int, initialized, rwo>
%g3:<int, initialized, rwo>

Figure 3.4 The Memory State at line 7.

into local safety preconditions and global safety preconditions. The local safety preconditions are conditions that can be checked using typestate information alone. The global safety preconditions will need to be verified via further analysis. The global safety preconditions include array bounds checks, address alignment checks, and null pointer dereference checks.

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 3.5.

ASSERTIONS	LOCAL SAFETY PRECONDITIONS	GLOBAL SAFETY PRECONDITIONS
%o2 *is the address of an integer array:* %o2 mod 4 = 0 %o2 ≠ NULL	e is readable; e is initialized; %g2 is writable; %o2 is followable and operable	*Array bounds checks:* %g2≥ 0 ∧ %g2 < 4*n* ∧%g2 mod 4 = 0 *Alignment and null-pointer checks:* (%o2 + %g2) mod 4 = 0 ∧ %o2 ≠ null

Figure 3.5 Assertions and Safety Preconditions for Line 7.

Because %o2 stores the base address of an integer array, it must be word aligned and non-null. Since the instruction loads the contents from e and stores it into the register %g2, the local safety preconditions state that the location e must

be readable and initialized, %g2 must be writable, and %o2 must be followable and operable. The global safety conditions verify that the array index %g2 is within the array bounds and that the address calculated by "%o2+%g2" is properly aligned and non-null.

3.4.4 Local Verification

The *local-verifi cation* phase checks the local safety preconditions. It performs a linear scan over the instructions in the untrusted code. In our example, it finds that the local safety preconditions are all true at line 7. We will describe the annotation phase and the local-verification phase of our analysis in more detail in Chapter 5.

3.4.5 Global Verification

The *global-verifi cation* attempts to verify the global safety preconditions using program-verification techniques. In the presence of loops, we use the induction-iteration method [84] to synthesize loop invariants.

To make the global-verification phase more efficient, this phase also incorporates a symbolic range analysis that propagates range information of registers. This allows the analysis to avoid using expensive program-verification techniques wherever the range analysis is sufficient to verify that an array access is within the bounds.

For our example, symbolic range analysis will find that at line 7 the lower bound of %g2 is zero and the upper bound is $4n$-4. This analysis verifies that there are no array out-of-bounds violations at line 7.

Without using the range analysis, we would have to use the induction-iteration method to synthesize a loop invariant for the loop in lines 6-11. In this case, to prove that at line 7 index %g2 is less than the array upper bound, i.e., %g2 < 4n, we need to prove %g3 is less than n at line 6. (Note that the size of an integer is 4 bytes, and %g2 at line 7 is computed from %g3 at line 6 by the `sll` instruction.) The induction-iteration method can automatically synthesize the loop invariant "n > %g3 $\wedge$ $n \geq$ %o1". This invariant implies that "%g3 < n" holds at line 6, which in turn implies that "%g2 < 4n" holds at line 7.

In Chapter 6, we will describe the global-verification phase of our analysis in more detail.

Chapter 4

Typestate Checking

We describe the second phase of our safety-checking analysis. This phase abstractly interprets the untrusted code to produce a safe approximation of the memory state at each program point. The safe approximation of memory state is described using an *abstract storage model*. This model represents a memory state as a total map from abstract locations to typestates. An abstract location summarizes one or more physical locations, and a typestate describes the properties of the values stored in an abstract location.

This chapter is organized into two parts. In the first part, we describe the basic typestate system that includes an abstract storage model, an abstract operational semantics for SPARC machine-language instructions, and a typestate-checking algorithm that propagates typestate information. The typestate system incorporates a subtyping relationship among structures and pointers. This allows our analysis to check the safety of untrusted machine code that implements inheritance polymorphism via physical subtyping.

In the second part, we describe several techniques that strengthen our basic typestate-checking analysis. These techniques include a way to summarize function calls, and a method to detect stack-allocated arrays. Summarizing function calls allow our analysis to stop at the trusted boundaries.

4.1 Typestate System

The safety-checking analysis is based on an *abstract storage model*. The abstract storage model includes the notion of an *abstract store* and *linear constraints*. (We will describe linear constraints in more detail in Section 6.2.)

An abstract store is given by a total map from *abstract locations* to typestates. By design, the domain of abstract stores is a finite domain. (In contrast, the concrete stores form an infinite domain: 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.) Thus, an abstract location summarizes a set of physical locations. An abstract location has a name, size, offset, alignment, and optional attributes r and w to indicate whether the location is readable and writable by the untrusted code.

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

A *typestate* records properties of the values stored in an abstract location. A typestate is defined by a triple <type, state, access>. We define a meet operation $\sqcap$ on typestates so that typestates form a meet semi-lattice. The meet of two typestates is defined as the meet of their respective components. We describe type, state, and access component of our typestate system in the next few sections.

4.1.1 Type

In a machine-language program, a register or memory location can be used to store values of different types at different program points. The typestate-checking algorithm used in the typestate-propagation phase is a flow-sensitive analysis that determines an appropriate typestate for each abstract location at each program point by finding the greatest fixed point of a set of typestate-propagation equations. The typestate system incorporates a notion of subtyping among structures and 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 supertype of the acceptable values).

4.1.1.1 Type Expressions.

The type component of our type system is based on the physical type system of Siff *et al* [76]. Figure 4.1 shows the language of type expressions used in the typestate system. Compared with the type system of Siff *et al*, our typestate system additionally includes (i) bit-level representations of integer types, (ii) top and bottom types that are parameterized with a size parameter, (iii) pointer into the

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) → *t*	*Function*
| T(n)	*Top type of n bits*
| ⊥(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 4.1 A Simple Language of Type Expressions.
t stands for type, and *m* stands for a structure 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++.

middle of an array, and (iv) alignment and size constraints on types (which is not shown in Figure 4.1).

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.

A bit-level representation of integers allow 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, 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 a naïve type system (such as that is described in [94]) treats the value in `%g2` as an unsigned integer. In contrast, with the bit-level integer types of Figure 4.1, 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)).

The type $t(n]$ denotes a pointer that points somewhere into the middle of an array of type t of size n. Introducing pointers into the middle of an array allow our analysis to handle array pointers with better precision. For example, consider a program that reads from the elements of an array by advancing a pointer that initially points to the base address of the array at each iteration of a loop. The static type of the pointer inside of the loop will be a pointer into the middle of the array. This preserves the fact the pointer points to some element of the array. In contrast, a naïve type system would conclude that the pointer is a pointer to the element type of the array. With the naïve type system, we would have to forbid pointer arithmetic that advances the pointer to point to another element of the array.

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

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

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

$$[\textit{Pointer}]\ \frac{t <: t'}{t\ \texttt{ptr} <: t'\ \texttt{ptr}} \qquad [\textit{Array}]\ \frac{}{\begin{array}{l}t\ [i] <: t\ (i] \\ t\ (i] <: t\ [0]\end{array}}$$

Figure 4.2 Inference Rules that Define the Subtyping Relation.

4.1.1.2 A Subtyping Relation.

We now introduce a notion of subtyping on type expressions, adopted from the *physical-subtyping* system of Chandra and Reps [14], which takes into account the layout of aggregate fields in memory. Figure 4.2 lists the rules that define when a type *t* is a *physical subtype* of *t'* (denoted by *t* <: *t'*). 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* $t_2 <: t_1$.

In Figure 4.2, the rules [*Top*], [*Bottom*], [*Ground*], [*Pointer*], and [*Array*] are our additions to the physical-subtyping system given in [14]. 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 type *t* if the type of the first member of the structure is a subtype of *t*. 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. Rule [*Pointer*] states if *t* is a subtype of *t'*, then *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, and that all array types whose element type is *t* is a physical subtype of *t* [0]. (Rule [*Array*] is a little crude. It states that the meet of two array types of different sizes will return an array of size zero. This could cause the typestate-checking analysis to lose precision when checking certain programs. We will outline a solution to this problem in Section 7.3.2.)

When we make use of this notion of subtyping in the safety-checking analysis (see Section 4.3), 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.)

```
struct Point {          struct ColorPoint {     void f(Point* p)
  int(0:1:31) x;          int(0:1:31) x;        {
  int(0:1:31) y;          int(0:1:31) y;          p->x++;
};                        uint(24:0:8) color;     p->y--;
                        };                      }
```

Figure 4.3 Subtyping Among Pointer Types.

Allowing subtyping among integer types, structures, and pointers allows the typestate-checking 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. In this case, the actual parameter is a physical subtype of the formal parameter. Figure 4.3 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`.

4.1.1.3 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 4.4 provides a concrete example. 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`

```
   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 4.4 Rule [*Pointer*] is unsound for flow-insensitive type checking in the absence of aliasing information.
(Assume the same type declarations as shown in Figure 4.3.)

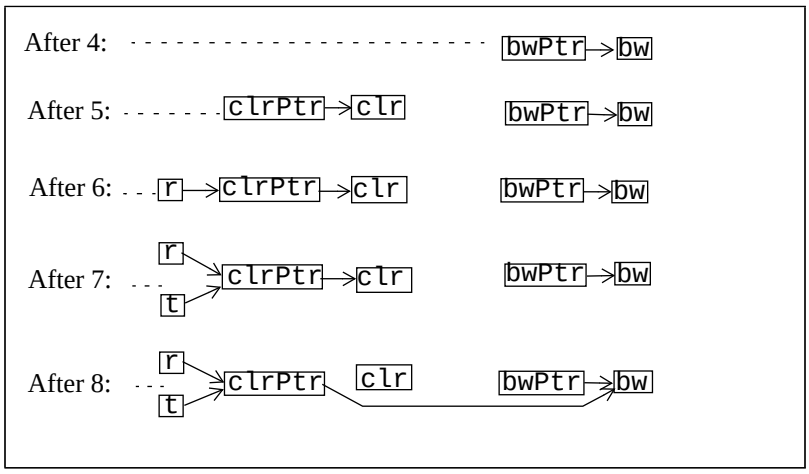


Figure 4.5 The contents of the store after each statement of function `f2` of Figure 4.4.

field at line 9. Figure 4.5 shows the contents of the store after each statement of function `f2`.

A static technique to handle this problem must 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 [88] avoids aliases altogether (and hence any indirect modifications) by " consum-

ing" a pointer as soon as it is used once. Smith *et al* [81] 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 any object that the field points to 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* [15], and are similar to the diagrams shown in Figure 4.5. We describe the state component of our typestate system in Section 4.1.2.) 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 4.4, one 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.

4.1.2 State

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

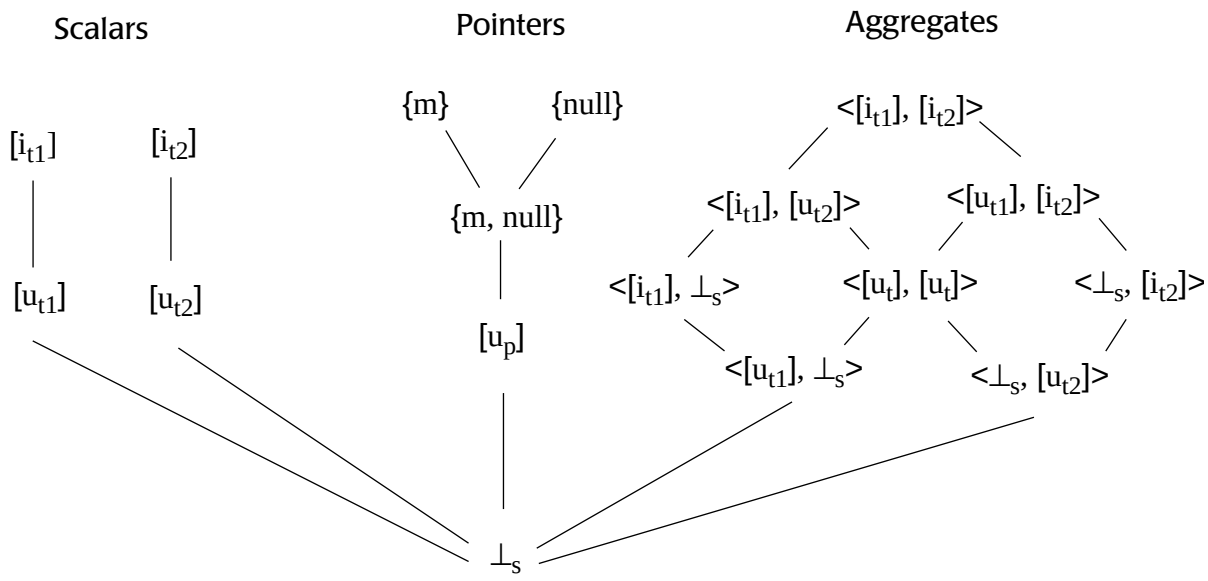


Figure 4.6 A Portion of the State Lattice.

state lattice contains a bottom element, denoted by $\perp_s$, that represents an undefined value of any type. Figure 4.6 illustrates selected elements of the state lattice. For a scalar type *t,* its state can be $[u_t]$ or $[i_t]$, which denote uninitialized and initialized values, respectively. For a pointer type *p*, its state can be $[u_p]$, which is the state of an uninitialized pointer, or *P*, a non-empty set of abstract locations referenced, where one of the elements of *P* can be *null*. For sets P_1 and P_2, we define $P_1 \leq P_2$ *iff* $P_2 \subseteq P_1$. For an aggregate type *G*, its state is given by the states of its fields. Since we also use the state descriptors to track abstract locations that represent pieces of stack- and heap-allocated storage, they resemble the storage-shape graphs of Chase *et al* [15].

4.1.3 Access Permissions

An access permission is either a subset of {*f*, *x, o*}, or a tuple of access permissions. If an abstract location stores an aggregate, its access permission will be a tuple of access permissions, with the elements of the tuple denoting the access permissions of the respective aggregate fields. The meet of two access-permission sets is their intersection. The meet of two tuples of access permissions is given by the meet of their respective elements.

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

In our model, a constant always has access permission *o*.

4.2 An Abstract Operational Semantics for SPARC 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.

Because machine-code operations are overloaded, the typestate lattice also includes a top element $\top$. This allows the typestate-propagation algorithm to perform overload resolution on-the-fly (see Section 4.2.1).

4.2.1 Overload Resolution

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

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

In the remainder of this section, we assume that we have non-`T` values at our disposal.

OPERATION	1 add $\mathbf{r}_s$, *Opnd*, $\mathbf{r}_d$	2	st $\mathbf{r}_s$, [$\mathbf{r}_a$+*n*]
ASSUMPTION	Scalar add	Array-index calculation $T(\mathbf{r}_s) = t\,[n]$	Store to an aggregate field Let $F = \{s.\beta \mid s \in S(\mathbf{r}_a),$ $\beta \in lookUp(T(s), n, 4)\}$
TYPE-PROPAGATION RULE	1. $T'(\mathbf{r}_d) = T(\mathbf{r}_s) \bigsqcap T(Opnd)$. 2. for $l \neq \mathbf{r}_d$, $T'(l) = T(l)$.	1. $T'(\mathbf{r}_d) = t\,(n]$. 2. for $l \neq \mathbf{r}_d$, $T'(l) = T(l)$.	1. if $F=\{l\}$, if l is not a summary location, $T'(l)=T(\mathbf{r}_s)$; otherwise $T'(l)=T(\mathbf{r}_s) \bigsqcap T(l)$. 2. if $\mid F \mid > 1$, for $l \in F$, $T'(l)=T(\mathbf{r}_s) \bigsqcap T(l)$. 3. for $l \notin F$, $T'(l) = T(l)$.
STATE-PROPAGATION RULE	1. $S'(\mathbf{r}_d) = S(\mathbf{r}_s) \bigsqcap S(Opnd)$. 2. for $l \neq \mathbf{r}_d$, $S'(l) = S(l)$.	1. $S'(\mathbf{r}_d) = S(\mathbf{r}_s)$. 2. for $l \neq \mathbf{r}_d$, $S'(l) = S(l)$.	1. if $F=\{l\}$, if l is not a summary location, $S'(l)=S(\mathbf{r}_s)$; otherwise $S'(l)=S(\mathbf{r}_s) \bigsqcap S(l)$. 2. if $\mid F \mid >1$, for $l \in F$, $S'(l)=S(\mathbf{r}_s) \bigsqcap S(l)$. 3. for $l \notin F$, $S'(l) = S(l)$.
ACCESS-PROPAGATION RULE	1. $A'(\mathbf{r}_d) = A(\mathbf{r}_s) \cap A(Opnd)$. 2. for $l \neq \mathbf{r}_d$, $A'(l) = A(l)$.	1. $A'(\mathbf{r}_d) = A(\mathbf{r}_s)$. 2. for $l \neq \mathbf{r}_d$, $A'(l) = A(l)$.	1. if $F = \{l\}$, if l is not a summary location, $A'(l)=A(\mathbf{r}_s)$; otherwise $A'(l) =A(l) \cap A(\mathbf{r}_s)$. 2. if $\mid F \mid >1$, for $l \in F$, $A'(l) =A(l) \cap A(\mathbf{r}_s)$. 3. for $l \notin F$, $A'(l) = A(l)$.

Figure 4.7 Propagation of Type, State, and Access information.

4.2.2 Propagation of Type, State, and Access Information

For the sake of brevity, Figure 4.7 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 $l \neq \mathrm{r}_d$ to denote $l \in (absLoc - \{\mathrm{r}_d\})$, and use $T(l)$ and $T'(l)$ to denote the types of abstract location l before and after the execution of an instruction. 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 $\emptyset$ if no such fields exist.

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

4.3 Typestate Checking

The typestate-propagation algorithm works on an interprocedural control-flow graph in which the nodes in the graph represent instructions and the edges represent control-flow in the usual fashion. For each instruction N (i.e., a node in the control-flow graph), we associate with it two total maps:

preTypestate(N): *absLoc*→typestate and
postTypestate(N): *absLoc*→typestate.

These two maps safely approximate the program states before and after each instruction at the respective program point. We define the meet of two total maps as the map that is constructed by performing a meet on their respective elements. That is, for m, n: $absLoc \rightarrow$typestate**,** $m \bigsqcap n$ is defined as follows: for any $s \in absLoc$, $(m \bigsqcap n)(s) = m(s) \bigsqcap n(s)$. To facilitate the presentation, we define the following notation:

- InitialStore: a map from abstract locations to the typestates for all abstract locations at the entry of the untrusted code. This initial store is computed in Phase 1.
- CFGedge: the set of all control-flow edges.
- Instruction: the set of all instructions in the untrusted code.
- StartNode: the instruction that is the entry point to the foreign code.
- Incoming, Outgoing: Instruction$\rightarrow$ 2^{CFGedge}, where 2^{CFGedge} denotes the power set of CFGedge. These two mapping functions give the control-flow edges that link an instruction to its control-flow predecessors and successors.
- Interpret: Instruction $\rightarrow_{\mathcal{R}}$ Interpret (N) gives the abstract operational semantics of the instruction N (see Section 4.2.2).

The algorithm is a standard worklist-based algorithm. It starts with the map $\lambda l.\top$ at all program points (i.e., all abstract locations have the typestate $\top$). The algorithm for typestate propagation is shown in Figure 4.8. Initially, the start node of the untrusted code is placed on the worklist. An instruction is chosen from the worklist and examined. If it is the start node, its *preTypestate* gets the *initialStore*. Otherwise, the typestates of the abstract locations at the entry of the examined instruction become the meet of the corresponding typestates at the exits of the instruction's predecessors. The instruction is interpreted abstractly

```
Typestate_Propagation() {
    worklist = {StartNode};
    while (worklist is not empty) {
        Select and remove an instruction N from worklist;
        If (N is the StartNode) {
            ts = InitialStore;
        } else {
            foreach (control-flow edge, <M, N> ∈ Incoming(N))
                ts = ts ⨅ postTypestate (M);
        }
        preTypestate(N) = ts;
        oldPostTypestate(N) = postTypestate(N);
        postTypestate (N) = Interpret (N) (preTypestate (N));
        if (postTypestate(N) ≠ oldPostTypestate (N)) {
            foreach (control-flow edge, <N,W> ∈ Outgoing (N))
                worklist = worklist ∪ { W };
        }
    }
}
```

Figure 4.8 Propagation of Typestate.
We use two nodes in the control-flow graph, e.g., <M,N>, to represent a control-flow edge. For the edge <M, N>, the node M represents the source of the edge and the node N represents the target of the edge.

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.

Figure 4.9 shows the results of typestate propagation applied to our running example. The right most column shows the instructions. The left most column shows the abstract store before the execution of the corresponding instruction.

TYPESTATE BEFORE	UNTRUSTED CODE
e: i_{int}, ro %o0: int[n], {e}, rwfo %o2: ⊥ %o1: i_{int}, rwo %g2: ⊥ %g3: ⊥	1: mov %o0,%o2
e: i_{int}, ro %o0: int[n], {e}, rwfo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: ⊥ %g3: ⊥	2: clr %o0
e: i_{int}, ro %o0: 0, rwo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: ⊥ %g3: ⊥	3: cmp %o0,%o1
e: i_{int}, ro %o0: 0, rwo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: ⊥ %g3: ⊥	4: bge 12
e: i_{int}, ro %o0: 0, rwo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: ⊥ %g3: ⊥	5: clr %g3
e: i_{int}, ro %o0: i_{int}, rwo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: ⊥ %g3: i_{int}, rwo	6: sll %g3, 2,%g2
e: i_{int}, ro %o0: i_{int}, rwo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: i_{int}, rwo %g3: i_{int}, rwo	7: ld [%o2+%g2],%g2
e: i_{int}, ro %o0: i_{int}, rwo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: i_{int}, rwo %g3: i_{int}, rwo	8: inc %g3
e: i_{int}, ro %o0: i_{int}, rwo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: i_{int}, rwo %g3: i_{int}, rwo	9: cmp %g3,%o1
e: i_{int}, ro %o0: i_{int}, rwo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: i_{int}, rwo %g3: i_{int}, rwo	10: bl 6
e: i_{int}, ro %o0: i_{int}, rwo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: i_{int}, rwo %g3: i_{int}, rwo	11: add %o0,%g2,%o0
e: i_{int}, ro %o0: i_{int}, rwo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: i_{int}, rwo %g3: i_{int}, rwo	12: retl
e: i_{int}, ro %o0: i_{int}, rwo %o2: int[n], {e}, rwfo %o1: i_{int}, rwo %g2: i_{int}, rwo %g3: i_{int}, rwo	13: nop

Figure 4.9 Results of Typestate Propagation.

Lines 6 to 11 correspond to the loop. Initially, %o0 holds the base address of the integer array ap (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).

In the next few sections, we describe several techniques that strengthen the typestate-checking analysis. These techniques include a technique to summarize calls to trusted functions, and a technique to determine types and sizes of stack-allocated arrays.

4.4 Summarizing 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 precondition, 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 preconditions describe the obligations that the actual parameters must meet, whereas the postconditions 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 is 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 [16, 17]. 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. Possible follow-on work to the thesis would be to 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 for short) 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 obligations as a list of pre-conditions of the form "*placeholder* : *typestate*".

The safety postconditions 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 postconditions of the form [*alias context, placeholder : typestate*]. An *alias context* [16] 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 preconditions, they represent additional safety require-

ments. When they appear in the postconditions, they provide additional information about the resulting memory state after the call.

```
int gettimeofday (struct timeval *tp);
  SAFETY PRECONDITION:
    %o0: <struct timeval ptr, {null, t}, fo>
    t: <struct timeval, u, wo>
  SAFETY POSTCONDITION:
    [(), 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 4.10 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.

To make this idea concrete, Figure 4.10 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 postconditions specify that after the execution of the call, the two fields of the location pointed to by `%o0` before the call will be initialized, and `%o0` will be an initialized integer. (Note that if `%o0` points to null before the execution of the call, the placeholder t becomes irrelevant because it would not be bound to any actual abstract location when we perform the *binding* process described latter in this section.) On a SPARC, the parameters to a function are passed through the out registers `%o0`, `%o1`, ..., `%o5`, and the return value of a function is stored in the register `%o0`.

In the example in Figure 4.10, the alias contexts were empty because there was 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 4.11, 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.

```
PointPtr g(PointPtr *p1, PointPtr* p2){
   *p2 = null;
   return *p1
}
```

SAFETY PRECONDITION:

`%o0`: <`PointPtr ptr`, {q1}, fo>

`%o1`:<`PointPtr ptr`, {q2}, fo>

q1: <`PointPtr`, {r1}, fo>

SAFETY POSTCONDITION:

[(q1 neq q2), `%o0` : <`PointPtr`, {r1}, ...>]

[(q1 eq q2), `%o0` : <`PointPtr`, {`null`}, ...>]

Figure 4.11 An example of safety pre- and post-conditions with alias contexts.

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 postconditions.

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.) This allows 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 postcondition. 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 contexts 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 postcondition. When the alias context

cannot be determined to be either definitely true or definitely false, the update specified by the postcondition may or may not take place. We make a safe assumption by performing a weak update.

4.5 Detecting Local Arrays

Determining type and bounds information for arrays that reside on the stack is difficult. 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 4.12 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 4.12*)*. 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 to either *SF*[96] or *SF*[104]. We now have a candidate for a local array. The reasoning runs as follows:

C PROGRAM	SPARC MACHINE LANGUAGE	FIRST TIME	SECOND TIME

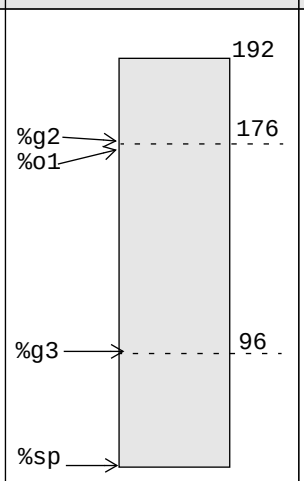
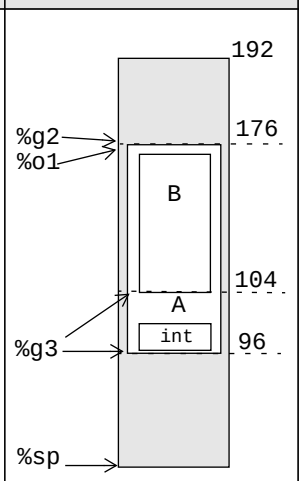
`typedef struct {`
`  int f;`
`  int g;`
`} s;`
 	
`int main() {`
`  s a[10];`
`  s *p = &a[0];`
`  int i=0;`
`  while (p<a+10) {`
`  (p++)->f = i++;`
`  }`
`}`
	`1:  main:`
	`2:  add %sp,-192,%sp`
	`3:  add %sp,96,%g3`
	`4:  mov 0,%o0`
	`5:  add %sp,176,%g2`
	`6:  cmp %g3,%g2`
	`7:  bgeu .LL3`
	`8:  mov %g2,%o1`
	`9:  .LL4:`
	`10:  st %o0,[%g3]`
	`11:  add %g3,8,%g3`
	`12:  cmp %g3,%o1`
	`13:  blu .LL4`
	`14:  add %o0,1,%o0`
	`15: .LL3:`
	`16:  retl`
	`17:  sub %sp,-192,%sp`
			

Figure 4.12 Inferring the Type and Size of a Local Array..
The label `.LL4` represents the entry of the while loop.

if we create two fictitious components *A* and *B* of *SF* (as shown in the right-most column in Figure 4.12), 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 (that is compatible with 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:

- 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.
- 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 (i.e., other than the pointer under consideration). The global-verification phase will verify later that all references to the local array are within the inferred bounds.

- 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 T(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 T. In this example, the τ that we find is

```
struct {
     int m1;
     T(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 bounds 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 consistent with the rest of the program. (ii) The global-verification phase will verify later that all references to the inferred 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 n-dimensional array involves a linear indexing scheme, which is reflected in linear relationships that the analysis infers for the values of registers.

4.6 Related Work

We compare our typestate-checking analysis with Mycroft's technique [52] that recovers type information from binary code.

Mycroft's technique reverse engineers C programs from target machine code using type-inference techniques. His type-reconstruction algorithm is based on Milner's algorithm W [54]; it associates type constraints with each instruction in an static single-assignment (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.

Chapter 5

Annotation and Local Verifi cation

The *annotation* phase of our safety-checking analysis annotates each instruction in a piece of untrusted code with safety preconditions and assertions. The safety preconditions assert that each instruction obeys the safety properties we enforce. The assertions are facts that can be used to assist the validation of the safety preconditions. The local-verification phase verifies the safety preconditions that can be validated using the results of typestate propagation alone. We describe the annotation and local-verification phases and illustrate them by means of our running example.

5.1 Annotation

The annotation phase consists of two steps: The first step attaches a collection of safety predicates to each usage of an instruction; the second step annotates each instruction in the untrusted code with safety preconditions and assertions via a linear scan of the untrusted code.

5.1.1 Attachment of Safety Predicates

For each different use of an instruction (e.g., a scalar add, or an add for array-index calculation), the annotation phase attaches to it a collection of safety predicates. The safety predicates assert that each instruction abides by the default safety conditions, and the host-specified access policy. The 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.

To illustrate how the local and global safety predicates are attached to each usage of an instruction, Figure 5.1 summarizes the safety predicates for two cases of `add` (scalar add and array-index calculation), one case of `st` that stores to an aggregate field, and one case of `ld` that loads from an array.

OPERATION	ASSUMPTIONS	LOCAL SAFETY PREDICATES	GLOBAL SAFETY PREDICATES
1 add $\mathbf{r}_s$, *Opnd*, $\mathbf{r}_d$	Scalar add	*operable*($\mathbf{r}_s$) $\wedge$ *operable*(*Opnd*)	
2 add $\mathbf{r}_s$, *Opnd*, $\mathbf{r}_d$	Array-index calculation T($\mathbf{r}_s$) = *t* [*n*]	*operable*($\mathbf{r}_s$) $\wedge$ *operable*(*Opnd*)	null $\notin$ S($\mathbf{r}_s$) $\wedge$ *inbounds*(*sizeof*(*t*), *0*, *n*, *Opnd*)
3 st $\mathbf{r}_s$, [$\mathbf{r}_a$+*n*]	Store to an aggregate field Let $F = \{s.\beta \mid s \in S(\mathbf{r}_a), \beta \in lookUp(T(s), n, 4)\}$	*followable*($\mathbf{r}_a$) $\wedge$ *operable*($\mathbf{r}_a$) $\wedge$ $F \neq \emptyset \wedge$ forall $l \in F$, *assignable*($\mathbf{r}_s$, *l*)	null $\notin$ S($\mathbf{r}_a$) $\wedge$ forall a $\in S(\mathbf{r}_a)$, *align*(*Align*(a)+n, 4) $\wedge$ *sizeof*(T($\mathbf{r}_s$))=4
4 ld [$\mathbf{r}_a$+*Opnd*], $\mathbf{r}_d$	Load from an array T($\mathbf{r}_a$) = *t* [*n*] T(*Opnd*) = int(0:s:v) S($\mathbf{r}_a$) = {e}	*followable*($\mathbf{r}_a$) $\wedge$ *operable*($\mathbf{r}_a$) $\wedge$ *operable*(*Opnd*) $\wedge$ *readable*(*e*) $\wedge$ *operable*(*e*) $\wedge$ *assignable*(*e*, $\mathbf{r}_d$)	*align*($\mathbf{r}_a$+n, 4) $\wedge$ *inbounds*(*sizeof*(*t*), *0*, *n*, *Opnd*) $\wedge$ *sizeof*(T($\mathbf{r}_d$))=4

Figure 5.1 Attachment of Safety Properties.
All registers are readable and writable by default.

- For scalar `add`, the safety predicate specifies that uninitialized values must not be used. The predicate *readable*(l) is true *iff* l is readable, and the predicate *operable* is true *iff* $o \in \mathrm{A}(l)$ and $\mathrm{S}(l) \notin \{[\mathrm{u}_{\mathrm{T}(l)}], \perp_{\mathrm{s}}\}$. The predicate *readable* does not appear explicitly in Figure 5.1 because all registers are readable and writable by default.

- The safety predicates for an array-index calculation state that $\mathbf{r}_{\mathrm{s}}$ and *Opnd* must both be readable and operable, and the index must be within the bounds of the array. The predicate *inbounds*($size$, low, $high$, i) is true *iff* $low \times size \leq i < high \times size$, *align*($i$, $size$). The predicate *align*(A, n) is true *iff* $\exists\ \alpha$ st. $A = n\alpha$.

- The safety predicates for `st` state that (i) $\mathbf{r}_{\mathrm{a}}$ must be followable and n must be a valid index of a field of the right size; (ii) $\mathbf{r}_{\mathrm{a}}$ must be non-null, and the address "$\mathbf{r}_{\mathrm{a}}$+n" must be properly aligned. The predicate *followable*(l) is true *iff* $f \in \mathrm{A}(l)$, and $\mathrm{T}(l)$ is a pointer type; the predicate *assignable*(m, l) is true *iff* *readable*(m), *writable*(l), and ($\mathrm{T}(l) \leq \mathrm{T}(m)$, *align*(*Align*($l$), *Align*($\mathrm{T}(m)$)) and *sizeof*($\mathrm{T}(m)$)$\leq$*Size*($l$)) all hold;

- The `ld` instruction loads from an array. The local safety predicates for an `ld` instruction state that (i) the base address of the array must be followable and operable; (ii) the operand that is used as the array index must be operable and of integer type; (iii) the abstract location e that is used as a surrogate for all elements of the array must be readable and operable, and the predicate *assignable*(e, $\mathbf{r}_{\mathrm{d}}$) must be true. The global safety predicate ensures that the loading address is properly aligned, and the array index is within the bounds of the array.

5.1.2 Generating Safety Preconditions and Assertions

Given the safety predicates, the annotation phase performs a linear scan through the machine instructions of the untrusted code. It generates safety pre-

conditions by instantiating the safety predicates (see Figure 5.1) with the actual operands to the instructions. The instantiated safety predicates are called local or global safety preconditions depending on whether a precondition is instantiated from a local or a global safety predicate. Besides the safety preconditions, the annotation phase also attaches assertions to each instruction to assist the later verification phases. The assertions are facts that can be derived from the results of typestate propagation. For machine code, a type means more than just a set of values and a set of valid operations defined on those values; it also includes physical properties, such as size and alignment constraints. If the typestate-propagation algorithm establishes that a location stores a valid address of a certain type, then that address must have a certain alignment property. The assertions are a means to give the verifier access to such information.

5.2 Local Verifi cation

The verification of the local safety preconditions is a purely local process. This phase generates an error message for each violation of the local safety preconditions.

5.3 An Example

We illustrate how the annotation and local-verification phases are applied in our running example that sums an integer array. The results of the annotation phase are shown in Figure 5.2.

UNTRUSTED CODE	ASSERTIONS	SAFETY PRECONDITIONS	
		LOCAL	GLOBAL
1:mov %o0,%o2	$\exists \alpha$ s.t. %o0 = $4\alpha \wedge$ %o0$\neq 0$	*operable*(%o0)	
2:clr %o0	$\exists \alpha$ s.t. %o0 = $4\alpha \wedge$ %o2$\neq 0$	*operable*(%o0)	
3:cmp %o0,%o1	$\exists \alpha$ s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0 \wedge$ %o0 $= 0$	*operable*(%o0) $\wedge$ *operable*(%o1)	
4:bge 12	$\exists \alpha$ s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0$		
5:clr %g3	$\exists \alpha$ s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0$		
6:sll %g3,2,%g2	$\exists \alpha$ s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0$	*operable*(%g3)	
7:ld [%o2+%g2],%g2	$\exists \alpha$ s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0$	*followable*(%o2)$\wedge$ *operable*(%o2) $\wedge$*operable*(%g2)$\wedge$ *operable*(e) $\wedge$ *readable*(e)	$\exists \alpha$ s.t.(%g2+%o2)=4α $\wedge$ 0 $\leq$ %g2 $<$ 4n $\wedge$ $\exists \alpha$ s.t. %g2 = 4α
8:inc %g3	$\exists \alpha$ s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0$	*operable*(%g3)	
9:cmp %g3,%o1	$\exists \alpha$ s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0$	*operable*(%g3) $\wedge$ *operable*(%o1)	
10:bl 6	$\exists \alpha$ s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0$		
11:add %o0,%g2,%o0	$\exists \alpha$ s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0$	*operable*(%o0) $\wedge$ *operable*(%g2)	
12:retl	$\exists \alpha$ s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0$		
13:nop	$\exists \alpha$ s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0$		

Figure 5.2 Safety Preconditions Produced by the Annotation Phase.

The assertions are shown in column 2 of Figure 5.2. At the entry of the untrusted code the register %o0 stores the base address of an integer array, hence, %o0 must be non-zero and word-aligned. That is, "%o0$\neq$0 $\wedge$ $\exists$ α s.t. %o0 = 4α". Similarly, given that typestate analysis has revealed that %o2 stores the base address of the array at program points 2 to 13, the annotation phase will generate the assertions "$\exists$ α s.t. %o2 = $4\alpha \wedge$ %o2$\neq 0$" for each of these points.

Columns 3 and 4 in Figure 5.2 show the local and global safety preconditions generated for each instruction in the untrusted code. For example, for the load instruction at program point 7, the typestate information for point 7 indicates that the register %o2 is the base address of an integer array, and %g2 is an integer that is used as an array index. (That is, the load instruction loads from an arbitrary element of the array and stores it into %g2.) The local safety preconditions state that %o2 must be readable, and the value stored in %o2 must be followable and operable. Because the abstract location e is used as a surrogate for all elements of the array, e must be both readable and operable. The global safety conditions at program point 7 state that the address calculated by "%o2+%g2" must be 4-byte aligned, and the value stored in %g2 that is used as an index must be greater than or equal to zero and less than the upper bound of the array. (The upper bound is $4n$ because the size of an array element is 4.) The global safety precondition also states that the array index %g2 should be 4-byte aligned.

The local-verification phase is a local process, and it generates an error message for each violation of the local safety preconditions. For the running example, it is able to verify that the local safety preconditions are all true.

Chapter 6

Global Verifi cation

The global-verification phase of the safety-checking analysis verifies the global safety preconditions, which check for array out-of-bounds violations, address misalignment violations, and null-pointer dereference violations. The global safety preconditions are represented as *linear constraints*. To verify the global safety preconditions, we take advantage of the synergy of an efficient *range analysis* and an expensive but powerful *program-verifi cation* technique that can be applied on demand. The range analysis determines safe estimates of the range of values each register can take on at each program point, which can be used for determining whether accesses on arrays are within bounds. We apply the program-verification technique only for safety preconditions that cannot be proven just by using the results of range analysis.

The rest of the chapter starts with an introduction to program verification, followed by a description of the linear constraints that we use to represent global safety preconditions, the technique we use to synthesize loop invariants, and the

symbolic range analysis that we use to perform array bounds checks. A key to automated program verification is that the system must be able to synthesize loop invariants. We describe the induction-iteration method for synthesizing loop invariants, and our extensions to the induction-iteration method.

6.1 Program Verifi cation

Floyd set the cornerstone of program verification with the idea of attaching assertions to statements to describe their behavior [36]. His technique is called the *inductive assertion* method. His approach models a program as a flowchart (control-flow graph) where the nodes represent program statements, and the edges represent the control flow. Logical assertions are attached to the control-flow edges. A program is correct with respect to the assertions, if it can be shown that assertions are true whenever control passes over the edges. Hoare [39], among others, refined this approach by developing a concise notation to represent the effect of a program construct in terms of a logical precondition and a postcondition. He also developed a set of axioms and inference rules for reasoning about these pre- and post-conditions.

Traditionally, program verification has focused on proving that a program conforms to its logical specification by giving a precondition that is true of program's inputs at the program's entry point and verifying that a postcondition is true at the exit. If we can attach assertions to the edges of the program's flowchart, and show that the assertions are all true whenever control passes over the edges,

then the postcondition is true along all paths from the program entry to the program exit.

For programs that have acyclic flowcharts, the attachment and the checking of the assertions can be done by (i) generating verification conditions (VCs), and (ii) verifying the VCs using a theorem prover. VC generation can be either forward or backward. In the backward approach, given a statement and a postcondition that needs to be verified, VC generation finds the weakest precondition such that if the precondition is true and the statement terminates then the postcondition is true. With a forward approach, VC generation finds the strongest postcondition of a program construct given the precondition.

A major difficulty with automated program verification is that the system needs to synthesize loop invariants in the presence of loops. For a program containing loops, a naïve VC generation process may not terminate due to the cycles in the program's control-flow graph. To cope with this problem, the program is partitioned at the loop entry points, and a *loop invariant* is synthesized for each loop. A loop invariant is an assertion such that if the assertion is true at the loop entry, it implies that it is also true when the loop is entered for each iteration. Thus, verifying the correctness of a program that contains a loop involves proving that (i) the loop invariant implies the postconditions to be proved, (ii) the loop invariant is true on entry to the loop, and (iii) the loop invariant is reestablished on each subsequent iteration.

Instead of proving that a program a program conforms to a specification in a general logic, which is difficult to achieve mechanically, the global-verification

phase of our analysis focuses only on a small set of conditions to prevent the program from performing out-of-bounds array references, misaligned loads and stores, and null pointer dereferences. In the next few sections, we will describe the linear constraints we use for expressing the global safety conditions, our theorem prover, and the induction-iteration method introduced by Suzuki and Ishihata [84] for synthesizing loop invariants. Unlike standard techniques for program verification, in which one monolithic VC is created containing all properties to prove, the induction-iteration method checks the validity of the global safety preconditions in a demand-driven fashion, and verifies the conditions one at a time.

6.2 Linear Constraints and Theorem Prover

Because 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 [44]. 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 [44]. Presburger formulas are decidable. Below, we give a few examples of linear constraints:

$0 \leq index \wedge index < Length$

$pointer < 0 \vee pointer > 0$

$\exists\ \alpha$ s.t. $address = 4\alpha$.

The first constraint expresses that an array index is within the bounds of an array of size *Length*. The second constraint expresses that a pointer is not null, and the third constraint expresses that an address is 4-byte aligned.

The Omega Library uses the Fourier-Motzkin method to determine if a Presburger formula is a tautology or is satisfiable. The basic operation used by the Omega Library is projection. Given a set of linear equalities and inequalities on a set of variables *V*, projecting the constraints onto the variables *V'* (where $V' \subset V$) produces a set of constraints on variables in *V'* that has the same integer solution as the original problem. The Omega Library determines if a set of constraints has integer solutions by using projection to eliminate variables until the constraints involve a single variable, at which point it is easy to check for an integer solution [44]. For example, projecting the set of constraints $x \leq y$ and $x+y \leq 1$ onto x generates $2x \leq 1$.

According to Pugh and Wannacott [68], if P and Q are propositions that can each be represented as a conjunction of linear equalities and inequalities, the Omega test can be used to check whether P is a tautology, whether P is satisfiable, and whether $P \supset Q$ is a tautology. Checking whether P is a tautology is trivial. For more details on the Omega Library, see Kelly *et al* [44] and Pugh *et al* [67,68,69].

6.3 Induction-Iteration Method

When the untrusted code contains loops, we need to synthesize loop invariants. In our system, the synthesis of loop invariants is attempted by means of the

induction-iteration method. We present the basic algorithm in this section, and describe our extensions to the induction-iteration method later in the chapter.

The induction-iteration method uses the " weakest liberal precondition" (wlp) as a heuristic for generating loop invariants. The weakest liberal precondition of statement S with respect to postcondition Q, denoted by wlp(S, Q), is a condition R such that if statement S is executed in a state satisfying R, (i) Q is always true after the termination of S (if S terminates), and (ii) no condition weaker than R satisfies (i). A weakest liberal precondition differs from a weakest precondition in that a weakest liberal precondition does not guarantee termination.

Since our technique works on machine language programs, we have extended the induction-iteration method to work on *reducible* control-flow graphs [61]. In the description below, we assume that the control-flow graph has been partitioned into code regions that are either cyclic (*natural loops*) or acyclic (see below).

A control-flow graph $G = (N, E)$ is reducible *iff* E can be partitioned into disjoint sets E_F, the *forward edge* set, and E_B, the *back edge* set, such that (N, E_F) forms a *DAG* in which each node can be reached from the entry node, and for each edge in E_B the target of the edge dominates its source [61]. Given a backedge $<m, n>$ (where m and n are control-flow graph nodes), the natural loop containing $<m, n>$ is the subgraph consisting of the set of nodes containing n and all the nodes from which m can be reached in the graph without passing through n, and the edge set connecting all nodes in this set [61]. In our implementation, we merge natural loops that share the same entry node to reduce the number of loops the global-verification phase has to analyze.

We believe our restriction to reducible control-flow graphs does not represent a significant limitation in practice. For example, a study cited by Muchnick [61] showed that over 90% of a selection of real-world Fortran 77 programs have reducible control-flow graphs. In addition, any irreducible control-flow graph can be transformed into a reducible one using a technique called *node splitting* [61].

INSTRUCTION TYPE	WEAKEST LIBERAL PRECONDITION	COMMENTS
`mov` *Opnd*, $\mathbf{r}_d$	$Q[\mathbf{r}_d \setminus Opnd]$	Move
`add` $\mathbf{r}_s$, *Opnd*, $\mathbf{r}_d$	$Q[\mathbf{r}_d \setminus (\mathbf{r}_s + Opnd)]$	Integer Add
`sll` $\mathbf{r}_s$, *Opnd*, $\mathbf{r}_d$	$Q[\mathbf{r}_d \setminus (\mathbf{r}_s \times 2^{Opnd})]$	Logical Left Shift
`cmp` $\mathbf{r}_s$, *Opnd*	$Q[icc \setminus (\mathbf{r}_s - Opnd)]$	Comparison
`ld [`$\mathbf{r}_a$ `+` *Opnd*`],` $\mathbf{r}_d$	$Q[\mathbf{r_d} \setminus ([\mathbf{r}_a + \text{Opnd}])]$	Load
`bg label`	Q	Branch

Figure 6.1 Weakest Liberal Preconditions for Sample SPARC Instructions with respect to the Postcondition Q.

The method for generating wlps for non-conditional instructions is the same as those for generating weakest preconditions [39]. Figure 6.1 lists the *wlp*s with respect to a postcondition Q for a few sample SPARC instructions. In Figure 6.1, $\mathbf{r}_s$ and $\mathbf{r}_d$ are registers, *icc* is the integer condition code, and *Opnd* is either an integer constant n or a register $\mathbf{r}$. The operations +, –, and × are integer addition, subtraction, and multiplication. $Q[\mathbf{r} \setminus e]$ denotes Q with all occurrences of $\mathbf{r}$ replaced with the expression e.

In the reminder of this chapter, we will use [*addr*] to denote the value stored in the memory location that is at the address *addr*. To simplify matters, we will

omit the sizes of the memory locations. For example, [$\mathbf{r}_a$+*Opnd*] represents the value stored in the address "$\mathbf{r}_a$+*Opnd*"; the weakest liberal precondition of Q with respect to the load instruction " **d** [$\mathbf{r}_a$+*Opnd*], $\mathbf{r}_d$" is Q with all occurrences of $\mathbf{r}_d$ replaced with [$\mathbf{r}_a$+*Opnd*]. We will show how to compute the weakest liberal precondition for the store instruction in Section 6.4.2.

To compute the wlp for a natural loop, we define W(*0*) as the wlp generated by back-substituting the postcondition Q to be proved until the entry of a loop is reached, i.e., W(*0*) = wlp(loop-body, Q), and define W(*i+1*) as wlp(loop-body, W(*i*)). The wlp of the loop is the formula $\bigwedge_{i\geq 0}$W(*i*). We show how to compute the weakest liberal precondition of a condition with repect to an acyclic code region in Section 6.4.1.

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

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

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

Their argument runs as follows [84]:

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

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

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

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

The induction iteration method, in essence, iterates the following steps: create the expression L(*j*) as the current candidate for the loop invariant; generate VCs for (**Inv.0**(j)) and (**Inv.1**(j)); and attempt to verify the VCs using a theorem prover. Figure 6.2 shows the basic induction-iteration algorithm taken from Suzuki and Ishihata (rewritten in pseudo code) [84]. The algorithm will perform at most *MAX_NUMBER_OF_ITERATIONS* induction steps. As discussed further in Section 6.6, we have found in practice, that for our situation it is sufficient to set *MAX_NUMBER_OF_ITERATIONS* to three.

```
1: Induction_Iteration() : SUCCESS | FAILURE {
2:      i=0; Create formula W(0); //Try L(-1)
3:      while (i < MAX_NUMBER_OF_ITERATIONS) {
4:           case (Theorem_prover((⋀_{i-1 ≥ k ≥ 0} W (k)) ⊃ W (i))) { //inv.1(i-1)
5:                true:          return SUCCESS;
6:                otherwise:{ //Try L(i)
7:                     case (Theorem_prover(wlp(<code-along-path-to-loop-entry>,W(i)))) {//inv.0(i)
8:                          true:          W(i+1)=wlp(loop-body, W(i));
9:                                         i=i+1;
10:                         otherwise:     return FAILURE;
11:                    }
12:               }
13:          }
14:     }
15:     return FAILURE;
16:}
```

Figure 6.2 The Basic Induction-Iteration Algorithm.

The reader may be puzzled why the algorithm first tests for **inv.1**(i-1), and then tests for **inv.0**(i). This is because the test **inv.0**(i-1) that matches the test for

inv.1(i-1) is performed in the previous iteration. In the case of L(*-1*), **inv.0**(-1) holds vacuously because L(*-1*)=$\wedge_{-1 \geq i \geq 0}$W($i$) = true.

6.4 Enhancements to the Induction-Iteration Method

We have made several enhancements to the basic induction-iteration algorithm to handle the SPARC instruction set, store instructions, nested loops, and procedure calls, and to make the global verification phase more precise and efficient.

6.4.1 Handling the SPARC Machine Language

In this section, we discuss the extensions that are needed to handle the SPARC instruction set, including delay slots, annulled instructions, and the use of condition codes.

To handle delay-slot instructions (which are also possibly annulled), we construct the control-flow graph so that the instruction in the delay slot is itself a basic block. This basic block follows the basic block in which the branch instruction is the last instruction but precedes the branch target. If the instruction that precedes the delay-slot instruction has the annul bit set, the respective delay-slot basic block is included only in the taken path. Otherwise the basic block of the delayed instruction is replicated and included in both the taken and fall through paths (see Figure 6.3).

To handle the use of condition codes, we model the SPARC condition code with a variable *icc* that holds the result of an appropriate arithmetic operation (e.g. for

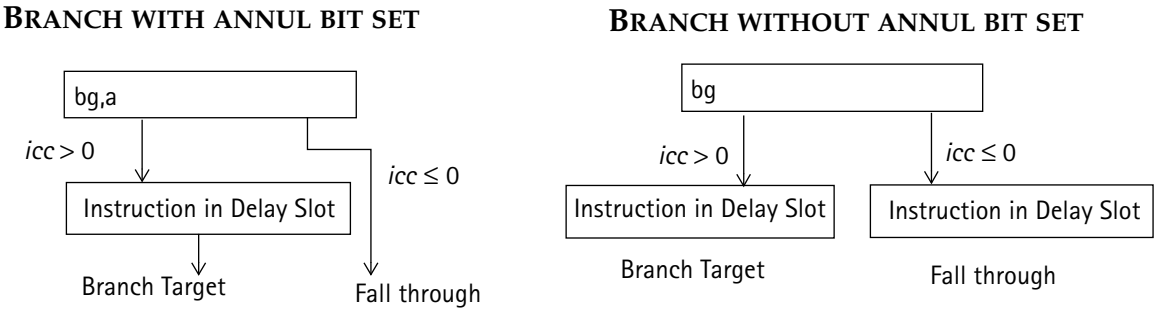


Figure 6.3 Branch With Delay Slot.

the instruction " cmp$\mathbf{r}_s$, *Opnd*", *icc* will hold "$\mathbf{r}_s$-*Opnd*"), and label its exit edges with conditions for the edges to be taken in the form of "*icc relop 0*". *relop* is a relational operator, including >, ≥, <, ≤, and =.

We illustrate this with the following code fragment with an annulled branch:

```
9: cmp %o0, %i2
10: ble,a target
11: add %o1, 1, %l0,
```

We use the line number (of an instruction) to represent an instruction, and a sequence of numbers to represent a path. Figure 6.4 illustrates how to compute the wlp of the condition Q (right before the program point 11) with respect to the path <9,10,11>. We model the effect of the " cmp"instruction as "*icc*=%o0-%i2", and label the edge that represents the fall through branch <10,11> with the condition {*icc* > 0}. We have wlp(<10,11>, Q)={*icc* > 0 ⊃ Q), and wlp(9, *icc* > 0 ⊃ Q) = {%o2-%i2> 0 ⊃ Q}.

To compute the wlp for an acyclic code region, the standard technique for verification-condition generation is used. Figure 6.5 illustrates this with an example where the condition for a branch to be taken in the indicated direction is enclosed

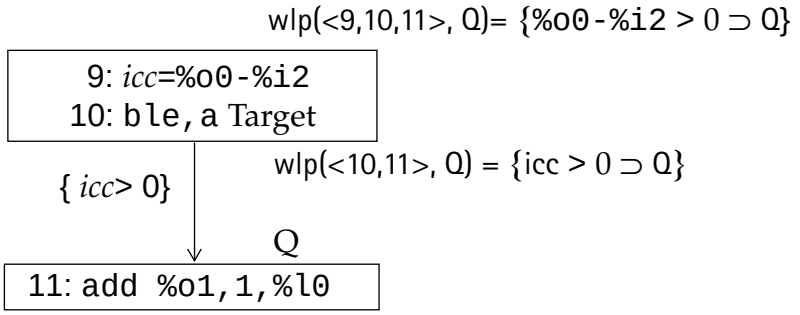


Figure 6.4 Handling SPARC Condition Code.

in curly brackets. During back-substitution, at a control-flow merge point that has multiple outgoing edges, the conjunction of the conditions for each of the outgoing edge of the junction node is computed first, before performing the back substitution across the junction node.

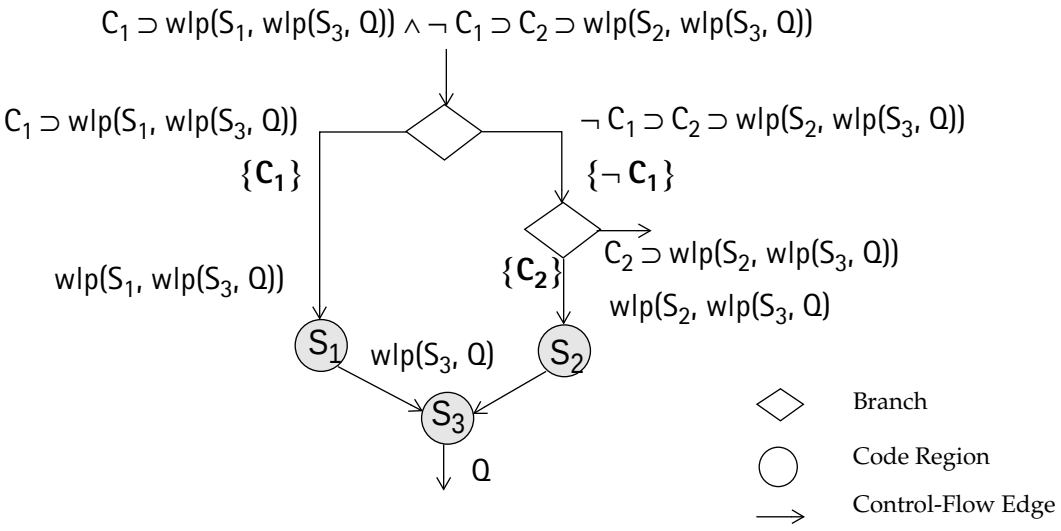


Figure 6.5 Weakest Liberal Precondition for an Acyclic Code Region.

6.4.2 Handling Store Instructions

We compute the wlp of store instructions based on Morris' general axiom of assignment [56], which provides a general framework for computing weakest precondition for assignments to pointer-typed variables.

We describe Morris' general axiom of assignment, and define the weakest liberal precondition for store instructions. Let p and q denote addresses, [p] denote a pointer dereference, and "st*e*, [p]" denote storing the result that is produced by evaluating the expression e into the memory location at address p. (We omit the sizes of memory locations.) We assume that e does not have side-effects. We define $\mathbf{[q]^{/p}_{e}}$ as follows:

•Definition 6.1: $\mathbf{[q]^{/p}_{e}} \equiv$ *if* $p=q$ *then* e *else* [q]

Let Q denote an arbitrary postcondition; the axiom for computing the wlp for a store instruction is

wlp("st*e*, [p]", Q) = Q [[q] \[q] $^{/p}_{e}$]

where Q [[q] \([q] $^{/p}_{e}$)] stands for Q with every occurrences of [q] in Q replaced by [q] $^{/p}_{e.}$ In other words, the weakest liberal precondition of the assignment is Q with each alias of [p] in Q replaced by e.

One difficulty with analyzing machine-language programs is that determining whether two addresses are the same cannot be determined, in many cases, until later during back-substitution. For example, consider the following code fragment that has both load and store instructions

```
1: mov %o1, %o2
```

```
2: st %i0, [%o0+%o1]
3: mov %o2, %o3
4: ld [%o0+%o3], %i1
5: {%i0=%i1}.
```

We want to verify the postcondition {%i0=%i1} at program point 5. Below we show the result of back-substituting the formula {%i0=%i1} across the four instructions in the above code fragment. As before, we use the line number (of the instruction) to represent the instruction.

wlp(4, {%i0=%i1}) = {%i0= [%o0+%o3]}

wlp(3, {%i0 = [%o0+%o3]}) = {%i0 = [%o0+%o2]}

wlp(2, {%i0 = [%o0+%o2]}) = $\{\%i0 = [\%o0+\%o2]\}^{/(\%o0+\%o1)}_{\%i0}$

= {if ((%o0+%o1)=(%o0+%o2)) then {%i0=%i0) else (%i0 = [%o0+%o2])}

wlp(1, $\{\%i0 = [\%o0+\%o2]\}^{/(\%o0+\%o1)}_{\%i0}$)

= {if ((%o0+%o1)=(%o0+%o2)) then {%i0=%i0) else (%i0 = [%o0+%o2])}

= {if ((%o0+%o1)=(%o0+%o1)) then {%i0=%i0) else (%i0 = [%o0+%o2])}

= {%i0=%i0} = true.

Note that the destination address of the store instruction at line 2 is the same as the source address of the load instruction at line 4. But, this cannot be determined until we back-substitute the formula across the mov instruction at line 1.

Having multiple store instructions in an instruction sequence further complicates the situation because we will not be able to determine the dependences among the instructions until the point at which we can disambiguate the addresses. A store instruction would affect a later load instruction only if it is not

killed by a later store instruction that is also before the load instruction. Morris' axiom of concurrent assignment handles this situation.

Let us consider a list of store instructions " st e_1, [p_1]; ...; st e_n, [p_n]" that is to be executed sequentially. We use $\underline{p}$ to denote the list of addresses "p_1, ..., p_n", and $\underline{e}$ to denote the list of expressions "e_1, ..., e_n".

For an arbitrary address q, we define

•Definition 6.2: $\mathbf{[q]}^{/\underline{p}}_{\underline{e}} \equiv$ *if* $(q = p_n)$ *then* e_n

else if $(q \neq p_n$ *and* $q = p_{n-1})$ *then* e_{n-1}

....

else if $(q \neq p_n$ *and ... and* $q \neq p_2$ *and* $q = p_1)$ *then* e_1

else if $(q \neq p_n$ *and ... and* $q \neq p_1)$ *then* [q];

With an arbitrary postcondition Q, the axiom for computing the wlp for a list of store instruction is

wlp(" st e_1, [p_1]; ..., st e_n, [p_n]", Q) = Q[[q] \($[q]^{/\underline{p}}_{\underline{e}}$)]

where Q[[q] \ ($[q]^{/\underline{p}}_{\underline{e}}$)] stands for Q with every occurrence of [q] in Q replaced by $[q]^{/\underline{p}}_{\underline{e}}$.

6.4.3 Handling Multiple Loops

The basic induction-iteration algorithm assumes that there is only a single loop in the program. To apply the induction-iteration method to programs with multiple loops, extensions are needed. To simplify matters, we show extensions to the basic induction-iteration algorithm for programs with two different kinds of

loop structures: (i) two consecutive loops, and (ii) two nested loops. All other cases are combinations of these two cases.

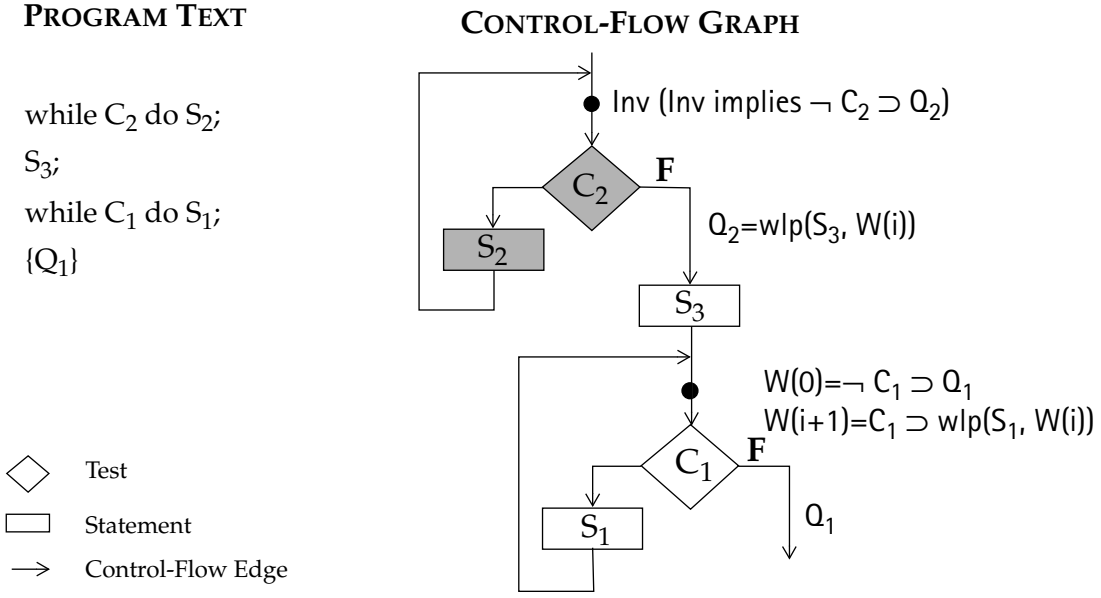


Figure 6.6 Applying the Induction Iteration Method to Two Consecutive Loops.

Figure 6.6 illustrates the steps to prove a postcondition Q_1 in a program with two consecutive loops. We assume that S_1, S_2, and S_3 themselves do not contain loops. To synthesize a loop invariant for the second loop that implies Q_1, in each induction step, we compute W(i) and establish that W(i) is implied by the conjunction of W(j) ($0 \leq j < i$), and that W(i) is true on entry to the second loop. To establish that W(i) is true on entry to the second loop, we may need to apply induction-iteration method recursively to the first loop. That is, we may need to synthesize a loop invariant for the first loop for each W(i) generated for the second loop. For example, to verify that W(i) is true on entry to the second loop, we

may need to synthesize a loop invariant Inv for the first loop, and Inv should imply the condition $\neg C_2 \supset Q_2$.

Figure 6.7 illustrates the steps needed to prove a postcondition Q_1 in a program with two nested loops. Again, we assume that S_1, S_2, and S_3 themselves do not contain loops. To compute W(i+1) for the outer loop from W(i), we need to use the induction-iteration method to synthesize a loop invariant Inv for the inner loop that has the property "$\text{Inv} \wedge \neg C_2 \supset \text{wlp}(S_3, W(i))$". Once Inv is synthesized, W(i+1) can be computed (in this case) using the formula $C_1 \supset \text{wlp}(S_2, \text{Inv})$. Moreover, given the way W(i+1) is obtained, W(i+1) being true on entry to the outer loop implies that Inv is true on entry to the inner loop. This indicates that there is no need to prove explicitly that Inv is true on entry to the inner loop. This makes synthesizing Inv local to the inner loop.

In our implementation, the extended induction-iteration algorithm also employs a backtracking strategy for finding W(i) at each induction step to overcome a problem that results from the weakening of the formula due to irrelevant conditionals in the program, and to apply a strategy called generalization (see Sectons 6.4.5 and 6.4.6). To use backtracking when synthesizing Inv, in the case of computing a wlp for an inner loop due to the synthesis of a loop invariant for an outer loop, instead of naïvely testing that W(i) of the inner loop is true on entry to the loop (a process that may never terminate), we record the current trial invariant L(j) of the outer loop, and first try to verify that L(j) implies W(i). If that fails, we test if W(i) is true on entry to the inner loop by disregarding all back edges along the paths from the entry of the program to the entry of the inner loop (i.e.,

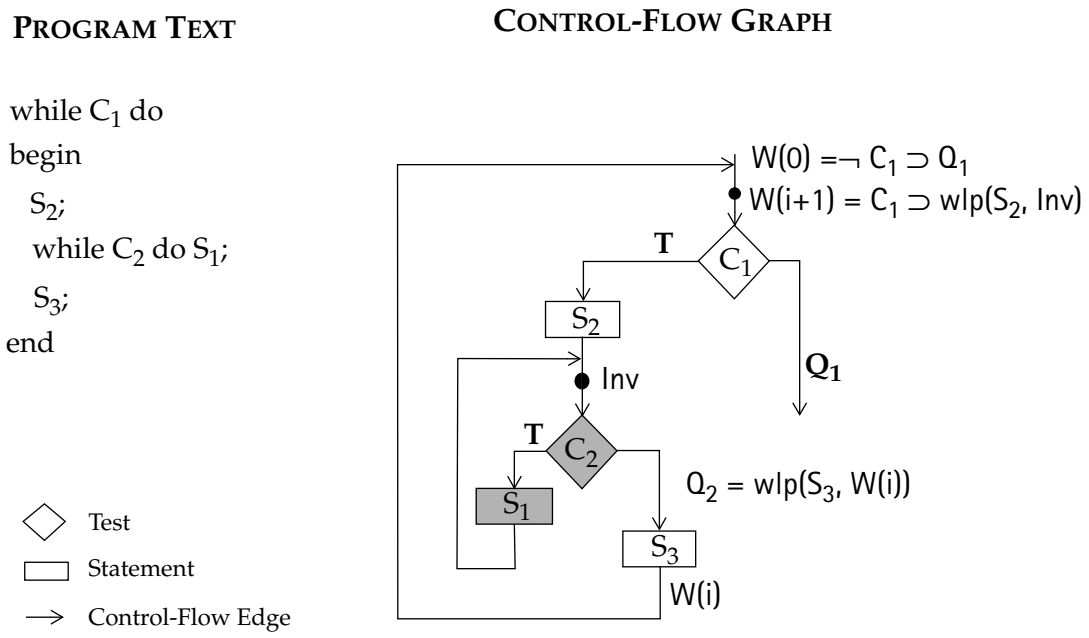


Figure 6.7 Applying the Induction Iteration Method to Two Nested Loops.

by treating the part of the program from the entry to the entry to the inner loop as if it were acyclic). These weak tests provide a means to drive the backtracking process when synthesizing invariants for an inner loop. This does not affect the soundness of our technique because each W(i) of the outer loop is computed by back-substituting the respective invariant synthesized in the inner loop, and our analysis will verify that each W(i) of the outer loop is true on entry to the outer loop.

6.4.4 Handling Procedure Calls

Procedure calls complicate the induction-iteration method in three ways: (i) handling a procedure call when performing a back-substitution, (ii) reaching the

procedure entry before we can prove or disprove a condition, and (iii) recursion. To handle procedure calls during back-substitution, we simply walk through the body of the callee as though it is inlined in the caller; this will generate a wlp for the callee function with respect to the postcondition that is propagated to the callsite. To make VC generation more efficient for procedural calls, we could do tabulation at procedure calls to reuse previous results of VC generation. When we reach the entry of a procedure, we check that the conditions are true at each callsite by using these conditions as postconditions to be proven at each of the caller. To simplify matters, our present system detects and rejects recursive programs. In principle, we could use the induction-iteration method to synthesize invariant entry conditions for recursive functions as we do for loops.

6.4.5 Strengthening the Formulae

Certain conditionals in the program can sometimes weaken L(j) to such an extent that it is prevented from becoming a loop invariant. For example, for the simple code fragment in Figure 6.8 that has two consecutive loops, we show how we fail to prove that the index variable i is greater than or equal to zero at program point 6. To verify {i ≥ 0} at program point 6, the induction-iteration method will produce {i< j ⊃ i ≥ 0} as $W_2(0)$ for the second loop. (We use a subscript to distinguish the W(i) produced for the first loop from those produced for the second loop.) To verify that $W_2(0)$ is true on entry to the second loop, we need to synthesize a loop invariant for the first loop. (To simplify things, we are ignoring the test at program point 3.) The $W_1(0)$ produced for the first loop is {i< j ⊃ i ≥ 0}, and

$W_1(1)$ is $\{i<(j+1) \supset i \geq 0\}$. The induction-iteration method will fail to synthesize a loop invariant for the first loop because $W_1(i)$ is strengthened after each induction step of induction iteration. For example, $W_1(0)$ is $\{i \geq j \vee i \geq 0\}$, whereas $W_1(1)$ is $\{ i \geq (j+1) \vee i \geq 0\}$, which is stronger than $W_1(0)$.

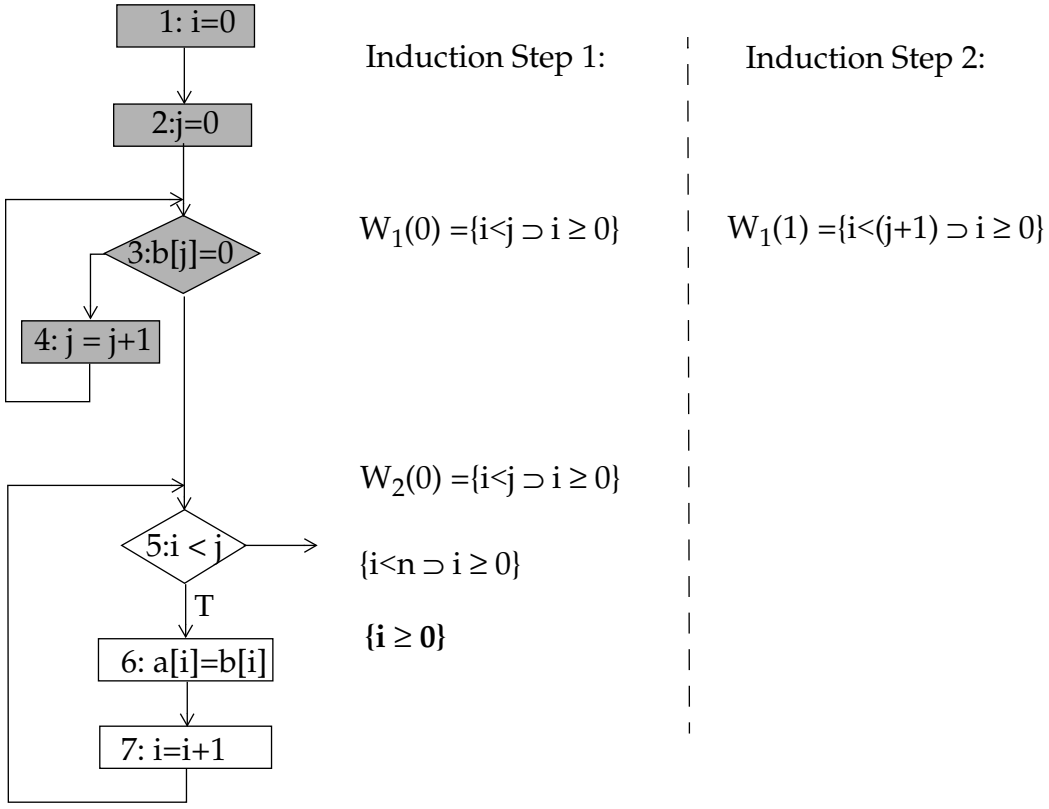


Figure 6.8 Conditionals in a program can sometimes weaken L(j) to such an extent that it is prevented from becoming a loop-invariant.

To address this problem, we strengthen L(*j*) by computing the disjunctive normal form of wlp(loop-body, W(*i-1*)), and trying each of its disjuncts as W(*i*) in turn. We rank the candidates according to a simple heuristic (by comparing the disjuncts with the original formula propagated to the same program point without

considering the conditionals) and test the potential candidates for W(i) using a breadth-first strategy. For the example shown in Figure 6.8, it is easy to verify that {i ≥ 0} (one of the disjunct of {i<j ⊃ i ≥ 0}) is a loop invariant for the first loop.

Figure 6.9 shows the extended induction-iteration algorithm with the breath-first search strategy. The algorithm uses a worklist to hold the candidates. Each item in the worklist has three components: i, f, and *prev*. The component f is the formula that is the tentative W(i). The component *prev* provides a backward link to obtain the respective W(k), (i-1 ≥ k ≥ 0) for the W(i) that is represented by f. Given a formula, the function *Get_Disjuncts* returns the set of formulae that are the disjuncts of the disjunctive normal form of the formula.

6.4.6 Incorporating Generalization

The breath-first strategy of the extended induction-iteration algorithm also incorporates a technique called *generalization,* also introduced by Suzuki and Ishihata [84]. The generalization of a formula f is defined as "¬(*elimination*(¬f))." Elimination uses the Fourier-Motzkin variable-elimination method to eliminate variables from the set of variables in ¬f to generate a simplified set of constraints that has the same integer solution as f. If there are several resulting generalizations, then each of them in turn is chosen to be the generalized formula. We will show how generalization is applied to our running example in Section 6.5.

6.4.7 Controlling the Sizes of the Formulae

The conditionals in a program can cause the formula under consideration to grow exponentially in size during verification condition generation. To reduce this

```
typedef struct node {
        i: int;
        f: Presburger_Formula;
        prev: node ptr
};
W:              Presburger_Formula [MAX_NUMBER_OF_ITERATIONS+1];
worklist:       list of node ptr;
Get_Disjuncts:  Presburger_Formula → set of Presburger_Formua

1: Induction_Iteration(Q) : SUCCESS | FAILURE {
2:      F = Get_Disjuncts(wlp(loop-body, Q));
3:      foreach (f ∈ F) worklist.append(node(0, f, NULL));
4:      while (worklist is not empty) {
5:          p = worklist.get();
6:          i=p.i;
7:          W[i] = p.f;
8:          np=p.prev; k=i-1;
9:          while (k ≥ 0) {
10:             W[k] = np.f; np=np.prev; k--;
11:         }
12:         case (Theorem_prover((⋀_{i-1 ≥ k ≥ 0} W (k)) ⊃ W (i))) { //inv.1(i-1)
13:             true:           return SUCCESS;
14:             otherwise:{ //Try L(i)
15:                 if (i < MAX_NUMBER_OF_ITERATIONS) {
16:                     if (Theorem_prover(wlp(<code-along-path-to-loop-entry>,W(i)))) {//inv.0(i)
17:                         F = Get_Disjuncts(wlp(loop-body, W(i)));
18:                         i=i+1;
19:                         foreach (f ∈ F) worklist.append(node(i, f, p));
20:                     } //if
21:                 } //if
22:             }
23:         }//switch
24      }//while
25      return FAILURE;
26:}
```

Figure 6.9 The Induction-Iteration Algorithm using Breath-first Search.

effect, back-substitution over a code region is performed in backwards topological order (with respect to the program's control-flow graph), and the formula at each junction point is simplified. This strategy effectively controls the size of the formulas considered, and ultimately the time that is spent in the theorem prover.

6.4.8 Sharing Common Computations

To reduce the number of times the induction-iteration algorithm is performed, we back-substitute all formulas to be proven until they reach a loop entry. We partition the formulae into groups that are made of comparable constituents, and invoke the induction-iteration algorithm starting from the strongest formulas in each group. If we can verify that a stronger formula is true, this implies that all formulae that are weaker than the formula are true.

6.4.9 Performing Simple Tests Assuming Acyclic Code Fragment

In a program that has many consecutive loops, to verify that each W(i) is true on entry to a later loop, we may need to synthesize loop invariants for each of the earlier loops. This must be done for conditions that eventually become part of the synthesized loop invariant, but it will be a waste effort for the conditions that will be rejected. To reduce this effect, for each candidate W(i), before we go into the expensive process of verifying that it is true on entry to the loop, we perform a simple test that treats the part of the program from the entry of the program to the entry of the loop as if it were acyclic. Performing this simple test allows us to handle a test case (a kernel device driver) that we cannot handle otherwise.

6.5 Example

Here we illustrate how the induction-iteration method is applied in our running example (first introduced in Figure 3.1 in Section 3.4). The control-flow graph of the program is shown in Figure 6.10. The instructions at lines 5 and 11 are replicated to model the semantics of delayed branches. We use a single integer variable icc to model the SPARC condition code and label each control-flow graph edge with the condition for that edge to be taken. Like before, 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.

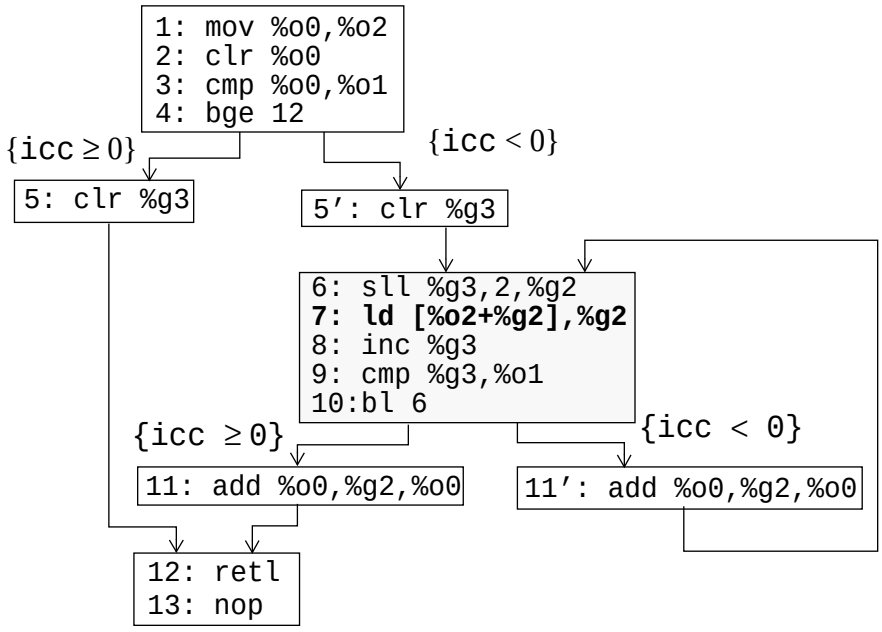


Figure 6.10 Induction Iteration: Example.

To verify that "%g2 < 4n" holds at line 7, we perform back-substitution, starting with "%g2 < 4n". Back-substituting this condition across line 6 produces "%g3

< n". Since the instruction at line 6 is the entry of a natural loop, we attempt to synthesize a loop invariant that implies "%g3 < n".

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

wlp(11', "%g3 < n") = "%g3 < n"

wlp(<11',10>, "%g3 < n") = "icc < 0 ⊃ %g3 < n"

wlp(<10,9>, "icc < 0 ⊃ %g3 < n") = "%g3 < %o1 ⊃ %g3 < n"

wlp(8, "%g3 < %o1 ⊃ %g3 < n") = "%g3+1 < %o1 ⊃ %g3+1 < n"

wlp(<7,6>, "%g3+1 < %o1 ⊃ %g3+1 < n") = "%g3+1 < %o1 ⊃ %g3+1 < n"

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

Unfortunately, W(0) ⊃ W(1) is not a tautology. Instead of continuing by creating W(2) etc., we strengthen W(1) using the generalization technique mentioned in Section 6.3. 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) $\wedge$ W(1) implies W(2).

By this means, the loop invariant synthesized for line 6 is "%g3 < n $\wedge$ %o1 $\leq$ n". This invariant implies that "%g3 < n" holds at line 6, which in turn implies that "%g2 < 4n" holds at line 7.

6.6 Scalability of the Induction-Iteration Method and Potential Improvements

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

One major cost of the verification phase is from performing the induction-iteration method, whose cost is determined by the number of iterations that have to be performed before an invariant is identified. The cost of an iteration step of the induction-iteration method is determined by the cost to perform VC generation and to invoke the theorem prover, and the cost to invoke the theorem prover is the dominate one. These costs are ultimately determined by the characteristics of the untrusted code. From our experience, it seems to be sufficient to set the maximum allowable number of iterations to three for each loop. The intuition behind this number is as follows: the first iteration will incorporate the conditionals in the loop into L(1), the second iteration will test if L(1) is already a loop invariant. No new information will be discovered beyond the second iteration.

Given that we limit the number of induction steps for each loop to three, and assume that there are *n* loops in the program, the worst-case scenario would

require 3^n induction steps to verify a safety condition. Consider a program that has n consecutive loops. The worst-case scenario happens if verifying that each W(i) is true on entry to a loop requires that we synthesize a loop invariant for its preceding loop; the same situation also happens at each preceding loop. The worst-case scenario could also happen for a program that contains n nested loops, if to compute a W(i+1) from a W(i) requires the synthesis of loop invariants for the inner loops. In practice, the worst-case scenario will seldom happen because the variables that are used in a loop are usually initialized right before the loop (hence, there is no need to go across the preceding loops to verify that a W(i) is true on entry to a loop). Furthermore, the tests in the inner loops usually do not contribute to the proof of a condition for an outer loop.

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

- The first would be to employ caching in the theorem prover: we can represent formulas in a canonical form and use previous results whenever possible.
- The second would be to perform tabulation at function calls or at nodes that have multiple incoming edges, and to reuse previous results of VC generation.
- The third would be to use more efficient algorithms for simple formulas. Several people have described methods that trade the generality of the constraint system for better efficiency. Bodik *at al* [10] describe a method to eliminate array-bounds checks for Java programs. Their method uses a restricted form of linear constraints called *difference constraints* that can be solved using an efficient graph-traversal algorithm on demand. Wagner *et al* [91] have formulated the buffer-overrun detection problem as *an integer constraint problem* that can be solved in linear time in practice.

- Finally, it might be profitable to use other invariant-synthesis methods in conjunction with induction iteration.

We chose to use the induction-iteration method to synthesize loop invariants because it works well for linear constraints and is totally mechanical. It is conceivable that we could use other techniques, such as the heuristic methods introduced by Katz and Manna [43] and Wegbreit [92], the difference equations method introduced by Elspas *et al* [25], or abstract interpretation using convex hulls described by Cousot and Halbwachs [21].

Cousot and Halbwachs' method works forward on a program's control-flow graph. It has the potential to speed up the induction-iteration method by pushing the facts forward in the program's control-flow graph. Our preliminary investigation demonstrated substantial speedups in the induction-iteration method by selectively pushing conditions involving array bounds forward in the program's control-flow graph. We will describe a symbolic range analysis for array-bounds checking in the next section. This symbolic range analysis is simpler than the method of Cousot and Halbwachs, but is more efficient.

6.7 Range Analysis

The induction-iteration method we have described in the previous sections and techniques such as those described by Cousot and Halbwachs [21] are quite powerful, but have a high cost. Here, we describe a simple range analysis that determines safe estimates of the range of values that each register can take on at each program point [86]. This information can be used for determining whether

accesses on arrays are within bounds. We take advantage of the synergy of the efficient range analysis and the expensive but powerful induction-iteration method. We apply the induction-iteration method only for the global safety pre-conditions that cannot be proven just by using the results of 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 $ax+by+c$ is because array-bounds checks usually involve checking either that the range of an array index is a subrange of [0, *length*-1], or that the range of a pointer that points into an array is a subrange of [*base*, *base*+*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 [$-\infty$, ∞]. 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

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 6.11 Binary Operations over Symbolic Expressions.

max is defined similarly except that it returns the greater of its two parameters, and ∞ if its two parameters are not comparable.

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, +, −, ×, and ÷, 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 6.11, 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

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	$[l_w, u_w]$	$[l_v, u_v]$
$w \leq v$	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 6.12 Dataflow Functions for Tests.

$\leq v$ and $w = v$ (where w and v are program variables). Figure 6.12 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 an 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.

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 a 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[1]), 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 utilize 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 $r' < n - c$.

6.7.1 An Example of Range Analysis

We now show how the range analysis is applied in our running example. In Figure 6.13, the left column shows the untrusted code that sums the elements of an integer array, the right columns show the results of range analysis. Each line

1. Although "while" and "for" loops are more common in source code, compilers typically transform them into 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".

UNTRUSTED CODE	PASS 1			PASS 2		PASS 3	
	%o1	%g2	%g3	%g2	%g3	%g2	%g3
1: mov %o0,%o2	[n,n]	[-∞,∞]	[-∞,∞]				
2: clr %o0	[n,n]	[-∞,∞]	[-∞,∞]				
3: cmp %o0,%o1	[n,n]	[-∞,∞]	[-∞,∞]				
4: bge 12	[n,n]	[-∞,∞]	[-∞,∞]				
5: clr %g3	[n,n]	[-∞,∞]	[0,0]				
6: sll %g3, 2,%g2	[n,n]	[0,0]	[0,0]	[0,4]	[0,1]	[0, 4n-4]	[0,n-1]
7: ld [%o2+%g2],%g2	[n,n]	[-∞,∞]	[0,0]	[-∞,∞]	[0,1]	[-∞,∞]	[0,n-1]
8: inc %g3	[n,n]	[-∞,∞]	[1,1]	[-∞,∞]	[1,2]	[-∞,∞]	[1,n]
9: cmp %g3,%o1	[n,n]	[-∞,∞]	[1,1]	[-∞,∞]	[1,2]	[-∞,∞]	[1,n]
10:bl 6	[n,n]	[-∞,∞]	[1,1]	[-∞,∞]	[1,∞]	[-∞,∞]	[1,∞]
11:add %o0,%g2,%o0	[n,n]	[-∞,∞]	[1,1]	[-∞,∞]	[1,n-1]	[-∞,∞]	[1,n-1]
12:retl	[n,n]	[-∞,∞]	[1,1]	[-∞,∞]	[0,∞]	[-∞,∞]	
13:nop	[n,n]	[-∞,∞]	[1,1]	[-∞,∞]	[0,∞]	[-∞,∞]	

Figure 6.13 Symbolic Range Analysis Applied to our Running Example.

shows the range of the registers after the execution of the instruction in the corresponding line. To simplify the presentation, we show only how the range information is propagated for three registers %o1, %g2 and %g3. (Recall that the base address of the array is passed to register %o0, and the size of the array is passed to register %o1.) Because the size of the array is n, and it is passed through %o1, the range of %o1 is [n, n] on entry to the untrusted code.

When the instructions are first visited, interpreting the clr instruction at line 5 sets %g3 to [0,0], interpreting the sll instruction at line 6 sets %g2 to [0,0], and interpreting the inc instruction in line 8 increments %g3 to [1,1]. Interpreting the branch instruction in line 9 sets the range of %g3 to [1,1] along the taken path. When the instructions are visited the second time, the range of %g3 is set to [0,1], which is the result of performing a meet of the range of %g3 propagated

along the edge <5, 6> and the range of %g3 propagated along the edge <11, 6>. Interpreting the inc instruction at line 8 sets the range of %g3 to [1, 2]. We perform a widening operation at line 9, and this brings the range of %g3 to [1, ∞]. (Note that [1,1] ∇ [1,2] = [1, ∞].) Interpreting the branch instruction at line 9 along the taken path sets the range of %g3 to [1, n-1]. In the third pass, the range of %g3 at line 6 becomes [0, n-1]. This is the result of performing a meet of the range [0,0] propagated along the edge <5,6> with the range [1,n-1] propagated along the edge <11, 6>. This causes the range of %g2 to be set to [0, 4n-4]. The algorithm then terminates because a fixed point has been reached. Knowing that the range of %g2 is [0, 4n-4] implies that %g2 is within the bounds of the array. This indicates that we do not have to use the induction-iteration method to prove that all array accesses are within bounds for this example. In Section 7.2, we show how the range analysis can actually speed up the global-verification phase for a few test cases.

Chapter 7

Experimental Evaluation

We present experimental evidence to demonstrate that our technique to enforce safe code execution is both feasible and practical. We describe a prototype implementation of our safety-checking analysis, and present our initial experience with it in which the prototype has been applied to a few case studies.

7.1 Prototype Implementation

All of the techniques described in the previous chapters, with the exception of the technique to infer sizes and types of local arrays (see Section 4.5), have been implemented in a prototype safety-checker for SPARC machine programs. The safety checker takes untrusted binary code that is in the form of executable or object files. It checks whether the untrusted code obeys the default safety conditions and the host-specified access policy, and reports reasons why it is unsafe if the code has safety violations.

Although the safety checker is implemented for SPARC machine language programs, the techniques embodied in it are essentially language independent.

As an alternative to reimplementing it for another language, it might be possible to employ binary translation techniques to achieve platform independence. That is, for untrusted code that is not written in SPARC machine language, we could first translate the untrusted code into semantically equivalent SPARC code before performing safety checking.

Figure 7.1 shows the flow chart of our prototype implementation. We recapitulate the five phases of our safety-checking analysis, and mention some details that have been left out of the previous chapters.

The preparation phase produces the *initial annotation* (the initial memory state at the entry of the untrusted code) and an interprocedural control-flow graph. To produce an interprocedural control-flow graph, our analysis identifies the boundaries of each function using symbol table information. Starting with the start function of the untrusted code, it dissembles each instruction, identifies the basic blocks, and builds intraprocedural control-flow graphs for all functions that are reachable from the start function. It identifies the targets of call instructions to build a static call graph. To find the targets of calls through registers, a simple intraprocedural constant propagation algorithm is performed. At the end, an interprocedural control-flow graph is produced based on the intraprocedural control-flow graphs and the static call graph. Note that the interprocedural control-flow graph could be incomplete. It will be refined during the typestate-propagation phase when typestate information becomes available. For example, with interface-based programming [51], the addresses of interface functions are typically stored in a structure. Calling an interface function typically involves a load

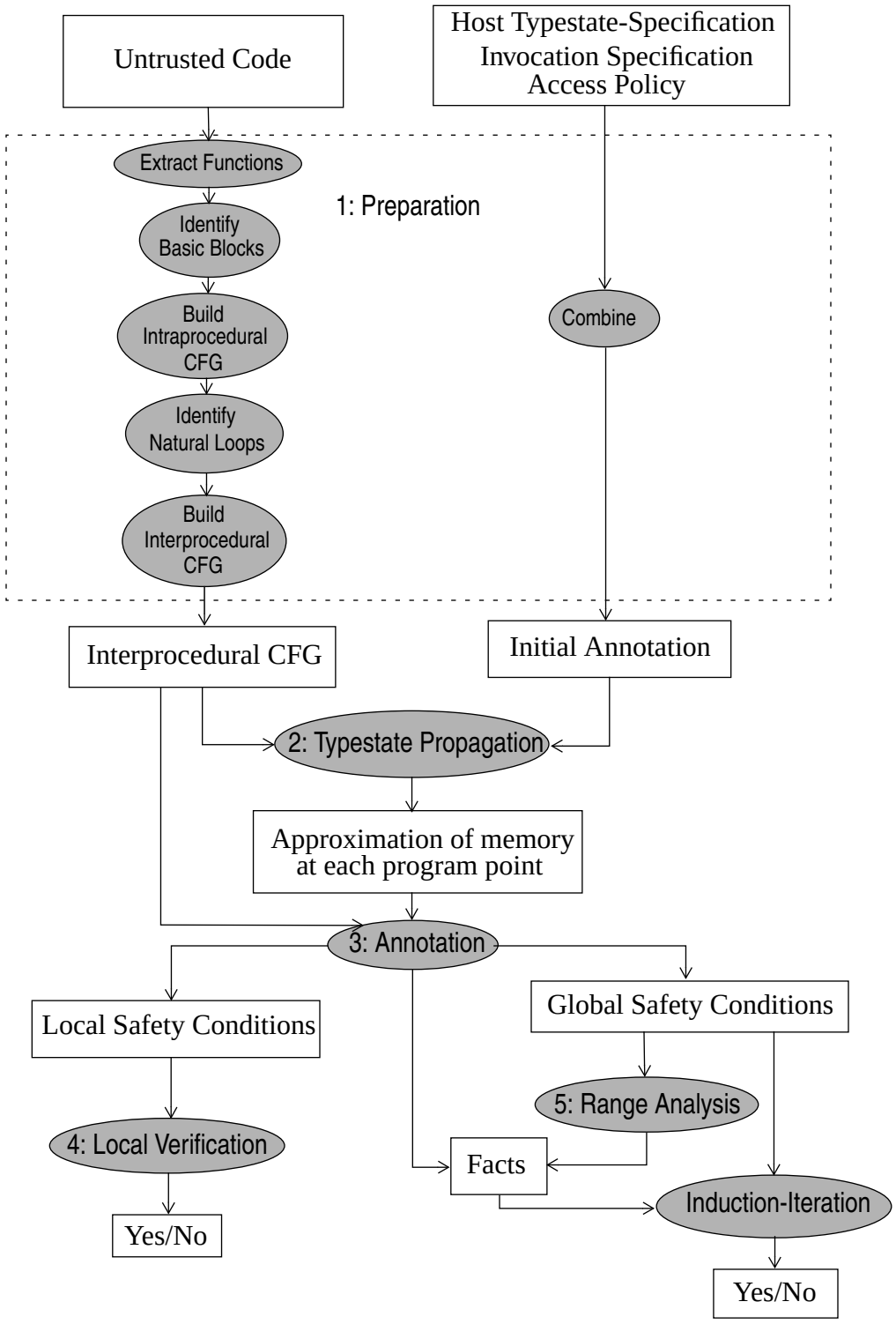


Figure 7.1 Prototype Implementation for SPARC Machine Language.

from the structure followed by a call through the register that holds the loaded value. The typestate of the loaded value, which provides the information about the callee, is not known before typestate-propagation is performed.

Besides the interprocedural control-flow graph and the initial annotation, the preparation phase also identifies natural loops and loop-nesting relationships. This information is used later by the global-verification phase. To identify the natural loops in a program, we implemented the algorithm developed by Lengauer and Tarjan for computing dominators and the algorithm Nat_Loop (both algorithms are described in Muchnick's book [61]). A node n in the control-flow graph is a dominator of a node m if all control-flow paths that reach node m from the entry of the function (that contains m and n) go through the node n. The algorithm Nat_Loop identifies a natural loop given a back edge (i.e., a control-flow edge whose target node dominates its source node). To reduce the number of natural loops that the global-verification phase has to analyze, our implementation merges natural loops whose back edges share the same target node. This is important because otherwise our analysis will have to treat them as nested loops, which are more expensive to analyze.

The typestate-propagation phase finds a safe approximation of the memory state at each program point. Our implementation replicates functions during the analysis to achieve some degree of context sensitivity: If the analysis finds that a parameter of a function could be pointing to arrays of different sizes, the function will be replicated so that in each copy the parameter points to only to arrays of one size.

For untrusted code that accesses structures that have multiple array-typed fields, we may need the bounds information of pointers or index variables to figure out to where a pointer points inside a structure. For this purpose, we run a simple range analysis on demand if such a situation arises. The simple range analysis is executed at most once for each procedure.

In our implementation, we have combined the annotation phase and the local verification phase to check the local safety conditions right after it is annotated.

The global-verification phase takes advantage of the synergy of range analysis and the induction-iteration method. We represent the results of range analysis as facts to be used by the induction-iteration method.

7.2 Case Studies

We now present our initial experience in which the system has been applied to a few case studies. The test cases we use include array sum (which has been used as the running example in previous chapters), start-timer and stop-timer code taken from Paradyn's performance-instrumentation suite [53], 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 [80], bubble sort, two versions of heap sort (one manually inlined version and one interprocedural version), stack-smashing (example 9.b described in Smith's paper [77]), MD5Update of the MD5 Message-Digest Algorithm [71], a few functions from jPVM [42], and the device driver /dev/kerninst [85] that comprises two modules: /dev/kerninst/symbol and /dev/kerninst/loggedWrites.

The stack-smashing program is a sample program that demonstrates how to take advantage of the vulnerability of the UNIX system; the intruder gains access to the system by overrunning the contents of the stack in a way that causes control to subsequently be transferred to the intruder's code. The MD5 Message-Digest Algorithm takes as input a message of arbitrary length and produces as output a 128-bit " figerprint" or " messagedigest" of the input. It is intended for digital-signature applications, where a large file must be " compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA. jPVM is a Java native interface to PVM for the Java platform. Java Native Interface (JNI) is a native-programming interface that allows Java code that runs inside a Java Virtual Machine to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembler [41]. In the jPVM example, we verify that calls into JNI methods and PVM library functions are safe, i.e., they obey the safety preconditions. /dev/kerninst is a device driver that can be installed in a Solaris kernel to allow the kernel instrumentation tool *kerninst* to modify a running kernel on-the-fly. The module /dev/kerninst/symbol provides the API for parsing the kernel symbol table, and the module /dev/kerninst/loggedWrites allows recoverable modifications to be made to (the instructions or data of) a run-time kernel. In our experimental study, we check the API function `kerninst_symtab_do` in the /dev/kerninst/symbol module, and the API function `loggedWrites::performUndoableAlignedWrites` in the /dev/kerninst/loggedWrites module. These are the major functions in the two modules. All examples are written in C or C++. All

	Sum	Paging Policy	Start Timer	Hash	Bubble Sort	Stop Timer	Btree	Btree2	Heap Sort 2	Heap Sort	jPVM	Stack-smashing	jPVM 2	/dev/kerninst /symbol	/dev/kerninst /loggedWrites	MD5
Instructions	13	20	22	25	25	36	41	51	71	95	157	309	315	339	358	883
Branches	2	5	1	4	5	3	11	11	9	16	12	89	16	45	36	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)	6	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)	48 (12)	6
Global Conditions (Bounds Checks)	4 (2)	9	13	15 (2)	16 (8)	17	35 (14)	39 (14)	56 (26)	84 (42)	49 (18)	100 (74)	99 (18)	116 (42)	192 (40)	121 (30)
Source Language	C	C	C	C	C	C	C	C	C	C	C	C	C	C++	C++	C

Figure 7.2 Characteristics of the Examples and Performance Results.

except the two /dev/kerninst example are compiled with gcc -O (version 2.7.2.3); the two dev/kerninst examples are compiled with Sun Workshop Compiler 5.0.

Figure 7.2 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 (total versus the number of bounds checks), and the source language in which each test case is written. We treat the checking of the lower and upper bounds as two separate safety conditions. A trusted function is either a host function or a function that we trust. We check that calls to a trusted function obey the corresponding safety preconditions of the function.

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-

	Sum	Paging Policy	Start Timer	Hash	Bubble Sort	Stop Timer	Btree	Btree2	Heap Sort 2	Heap Sort	jpvm	Stack-smashing	jpvm 2	/dev/kerninst /symbol	/dev/kerninst /loggedWrites	MD5
Typestate Propagation	0.02	0.05	0.02	0.04	0.04	0.03	0.09	0.11	0.17	0.15	0.63	0.69	3.05	4.88	15.4	5.92
Annotation	0.003	0.005	0.005	0.006	0.005	0.007	0.008	0.01	0.015	0.015	0.034	0.03	0.069	0.068	0.26	0.082
Range Analysis	0.01	0	0	0.01	0.03	0	0.03	0.04	0.08	0.12	0.13	0.54	0.24	0.68	0.95	1.24
Induction-Iteration	0.08	0.18	0.13	0.40	0.18	0.14	0.40	0.35	1.15	2.46	0.78	12.74	1.55	8.60	12.33	3.41
TOTAL (Seconds)	0.1	0.23	0.16	0.46	0.26	0.18	0.53	0.51	1.42	2.75	1.57	14.0	4.91	14.2	28.94	10.65

Figure 7.3 Performance Results.

smashing example and the symbol table module of /dev/kerninst. Figure 7.3 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 program verification using the induction-iteration method. The time to check these examples ranges from about less than a second to about 30 seconds.

The performance results for the /dev/kerninst/loggedWrites example were obtained based on a preliminary implementation of two new features (see Section 7.3.2 and Section 7.3.3 for the details). These features allow the analysis to handle the meet of array types of different sizes with better precision, and to account for the correlations among the induction variables in a loop when performing array bounds checks. Without these new features, the analysis would generate some false warnings about array out-of-bounds violations when check-

ing the /dev/kerninst/loggedWrites example. The /dev/kerninst/loggedWrites example takes longer time to check than the other test cases largely because the implementation of the new features is very preliminary with a goal to minimize the changes made to the existing infrastructure. The implementation was not sufficiently complete for a full test of its abilities to be made. Also note that, besides /dev/kerninst/loggedWrites, stack-smashing and /dev/kerninst/symbol also take longer times to check than the other examples. This is because that they have array out-of-bounds violations. We have to exhaust the breadth-first search of the induction-iteration method before we can conclude that an array out of-bound violation is possible.

We now highlight some of the benefits that our typestate system and the symbolic range analysis provide. 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 allows the analysis to use summaries of several host and library functions in hash, start- and stop timer, Btree2, the two jPVM examples, and the /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 [41] 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.

Symbolic range analysis allows the system to identify the boundaries of an array that is one field of a structure in the MD5 example. (We run an intraprocedural version of the range analysis on demand, when the typestate-propagation algorithm needs information about the range of a register value. The intraprocedural range analysis is run at most once for each function.) In the 11 test cases that have array accesses, range analysis eliminated 55% 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 -4% to 53% (with a median of 29%). Furthermore, in conjunction with simple test that treats part of the untrusted code as acyclic code (described in Section 6.4.9), range analysis allows us to verify the /dev/kerninst/symbol example, which we were not able to handle otherwise.

Figure 7.4 shows the times for performing program 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 not speeded up is because most array accesses in that example are out of bounds. When array accesses are actually out of bounds, range analysis will not speed up overall analysis because the analysis

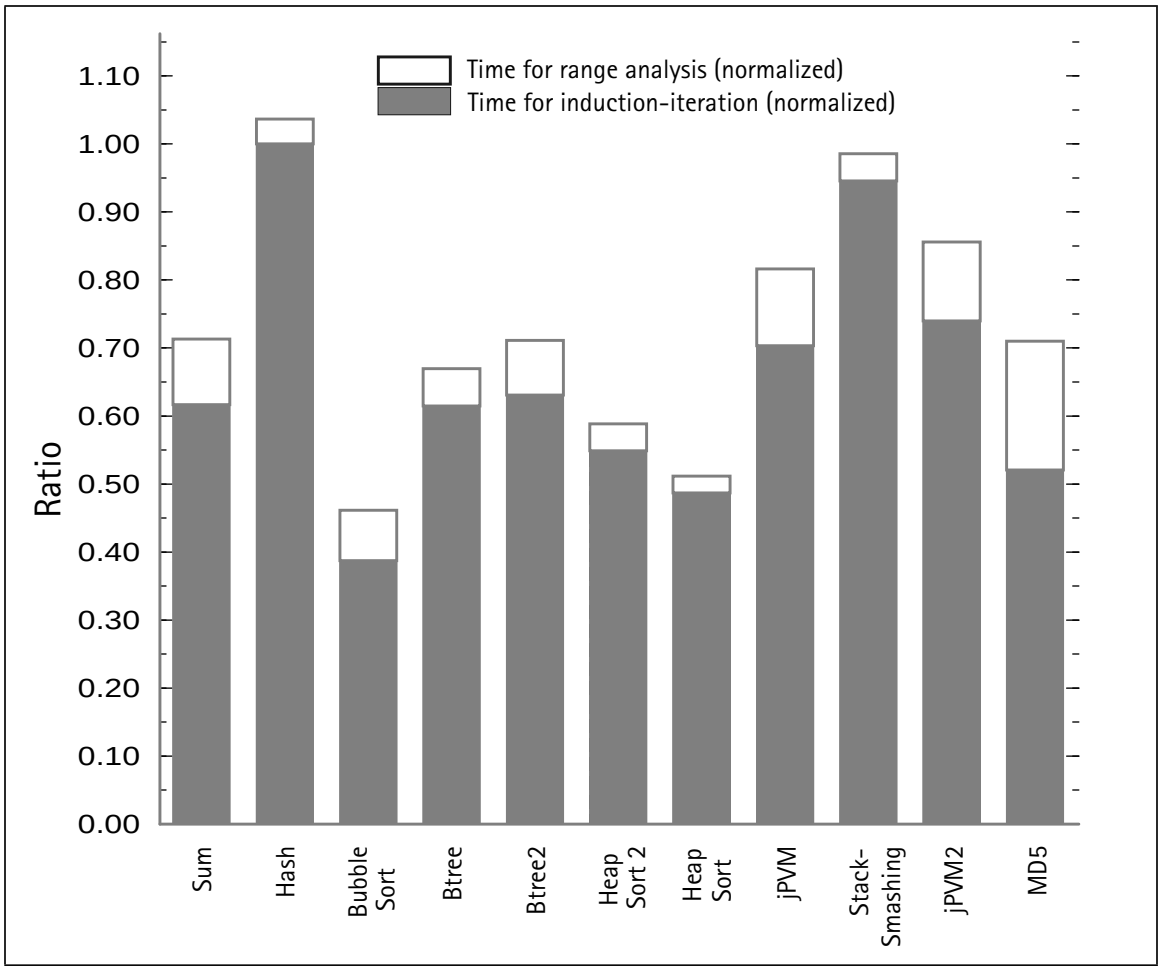


Figure 7.4 Times to perform global verification with range analysis normalized with respect to times to perform global verification without range analysis.

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 for array-bounds checking, and range analysis cannot prove that the array accesses are within bounds.

Note that range analysis has eliminated the need to synthesize loop invariants for array bounds checks, in about 55% of the cases. Two of the reasons why range analysis has not been able to do better are: (i) lost precision due to widening, and (ii) the inability of the range analysis algorithm to recognize certain cor-

relations among the registers. In our implementation of range analysis, we perform a widening operation just before the test to exit a loop for better precision. However, with nested loops, a widening operation in an inner loop could cause information in its outer loop to lose precision. A potential improvement to range analysis would be to not perform widening for variables that are invariants in the loop that contains the widening point. Another potential improvement is to identify correlations among loop induction variables and to include a pass after range analysis to make use of these correlations.

7.3 Limitations

Despite the initial success, our analysis has several limitations. We describe these limitations and outline possible solutions to them.

7.3.1 Lost of Precision due to Array References

The analysis may lose precision due to array references. Recall that we use a single abstract location to summarize all elements of an array, and model a pointer to the array base (or to an arbitrary array element) as a pointer that may point to the summary location. Our analysis loses precision when we cannot determine whether an assignment kills all elements of an array. For example, our analysis reported that some actual parameters to the host methods and functions are undefined in the jPVM examples, when they were in fact defined. We believe that dependence-analysis techniques such as those used in parallelizing compilers can be used to address this limitation [6,12].

7.3.2 Lost of Precision due to the Type System

The typestate-checking system described in this dissertation is not sophisticated enough to handle certain test cases. In particular, the subtyping rule states that the meet of two array types of different sizes will return an array of size zero. This rule will cause our analysis to lose precision when checking the safety of a variable-length vector type that is used in the /dev/kerninst/loggedWrites example. The vector type is implemented as a C++ template class shown below.

```
template<class T>
class vector {
   T* data_;
   int sz_;
   int tsz_;
   vector<T>& operator += (const T&);
   ...
};
```

The `data_` field of `vector` is an array, and its size is given by the field `tsz_`. The field `sz_` points to the next available slot in the vector. The operator "`+=`" adds an additional element into the vector. The C++ code that implements the operator "`+=`" (simplified) is illustrated in Figure 7.5.

At line 11, the type of `data_` is the meet of `T[tsz_]` and `T[newtsz]`, which is `T[0]`. For the array reference at line 11, our analysis will generate a false warning for the array bounds checks.

We sketch an enhancement to the typestate-propagation algorithm to address this limitation. The basic idea is to introduce a ϕ node for each meet of array types of different sizes, and propagate information about how each ϕ node is computed. We illustrate the technique with an example in Figure 7.6. The variables *a*

```
1: vector<T>::operator+=(const T &t) {
2:    const unsigned newsz = sz_ + 1;
3:    if (newsz > tsz_) {
4:       unsigned newtsz = unsignedint_bigger_than_newsz;
5:       T* newdata = new T[newtsz];
6:       copy(newdata, data_, sz_);
7:       delete data_;
8:       data_ = newdata;
9:       tsz_ = newtsz;
10:   }
11:   data_[sz_] = t;
12:   sz_++;
13:   return *this;
14:}
```

Figure 7.5 The operator "+=" of the Vector Type.

and b are of type T[n] and T[m]. Initially, p points to the base address of a at program point 0. The first time program points 1 and 3 are visited, p is of type T[n]. The first time program point 4 is visited, we introduce a ϕ node and a new variable w at program point 4 because of the meet of array types T[m] and T[n]. We record that w is either m or n depending on which edge was used to arrive at program point 4. The second time program point 1 is visited, we introduce another ϕ node and a new variable y. We record that y is either n or w. The second time program point 4 is visited, we update information about how w is computed. This process will eventually converge because for each array typed variable at each static program point we introduce at most one new variable. We can record this information on the control-flow edges to assist the global-verification phase (see the assignments in bold face attached to the control-flow edges). For example, if we were to check the condition "$i<w$" at program point 4, it will become "$i<m$"

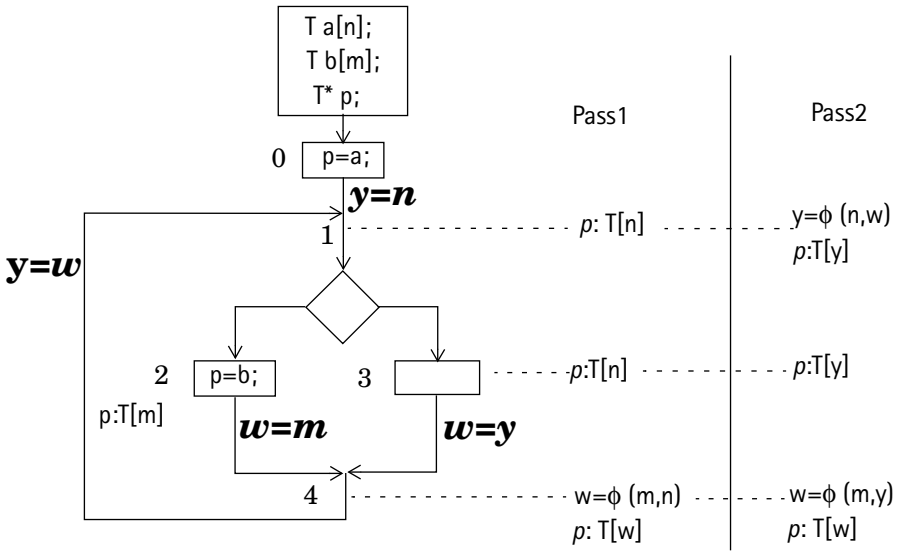


Figure 7.6 Introducing and Propagating ϕ nodes.

when we back-substitute the condition across the edge <4,2>. Similarly, it will become "*i*<*y*" if we back-substitute the condition across the edge <4,3>.

To perform the global verification for the `vector` type, we also need to know that the field `data_` is an array of size `tsz_` and that the value of the field `sz_` is less than or equal to the value of the field `tsz_`. This invariant can be represented by introducing symbolic names *m* and *n* (where 0 < n and 0 < m ≤ n), and encoded as the range information shown below:

```
data_ : T[n] // data_ is an array of type T and size n
tsz_ :[n,n] // tsz_ equals to n
sz_: [m,m] //sz_ equals to m.
```

7.3.3 Limitations of the Induction-Iteration Method

The induction-iteration method is not powerful enough to handle cases when the correlation between the loop tests and the calculation of an array index is not

```
// n>0
T a[n];
ptr = a;
x = n;
while ( (x--) > 0) {
   *(ptr++) = ....;
}
```

```
// m ≤ n, n>0
T a[n];
int x=0;
int y=0;
while(x<m) {
   a[y] = ...;
   x++,y++;
}
```

Figure 7.7 Two examples that the induction-iteration method cannot handle.

obvious. This can happen because of compiler optimizations such as strength reduction and optimizations to address-calculations. It can also appear in source code because of the programming style. We show two examples in Figure 7.7. Both examples appear in the implementation of the vector class that were described earlier.

The key to solving this problem is to uncover the correlations between loop tests and array-index calculations. A simple solution is to introduce a basic induction variable, and to represent all other induction variables as linear expressions of the basic induction variable. For the two examples in Figure 7.7, we can introduce a basic induction variable i and transform the examples into the corresponding programs shown in Figure 7.8. We now show how induction-iteration can handle the examples after they have been transformed.

To prove the condition "`ptr < a+n`" at line 6 in the program on the left-hand side of Figure 7.8, we have W(0) = {x>0 ⊃ ptr <a + n}. The fact that W(0) is true on entry to the loop can be shown by back-substituting W(0) across the path <4,3,2,1,0>.

```
0:// n>0                          0:// m ≤ n, n>0
1: T a[n];                        1:  T a[n];
2: int i=0;                       2:  int i=0;
3: T*ptr = a+i;                   3:  int x=i;
4: int x = n-i;                   4:  int y=i;
5: while (x > 0) {                5:  while(x<m) {
6:    *(ptr) = ....;              6:    a[y] = ...;
7:    i=i+1;                      7:    i=i+1;
8:    x = n-i;                    8:    x=i,y=i;
9:    ptr = a+i;                  9: }
10: }
```

Figure 7.8 After Introducing a Basic Induction Variable for the two examples shown in Figure 7.7.

Below we show the steps to compute W(1) by performing back-substitution of W(0) across the loop body:

wlp(9, {x>0 ⊃ ptr <a + n}) = {x>0 ⊃ a+i <a + n}

wlp(8, {x>0 ⊃ a+i <a + n}) = {n-i>0 ⊃ a+i <a + n} = {n-i>0 ⊃ i < n} = true

wlp(<7,6,5>, true) = true

Hence, W(1) is true, and the loop invariant synthesized is {x>0 ⊃ ptr <a + n}.

Verifying the program on the right-hand side of Figure 7.8 is slightly more complicated. To prove the condition " y < n" at line 6, we have

W(0) = {x<m ⊃ y < n}

Like the previous example, the fact that W(0) is true on entry to the loop can be shown by back-substituting W(0) along the path <4,3,2,1,0>.

Since that W(0) is not a tautology, we compute W(1) as follows:

wlp(8, {x<m ⊃ y < n})={i<m ⊃ i < n}

wlp(<7,5,6,5>, {i<m ⊃ i < n}) = {x<m ⊃ (i+1<m ⊃ i+1 < n)}

W(1) = {x<m ⊃ i+1<m ⊃ i+1 < n}

Instead of continuing to compute W(2), we strengthen W(1) by carrying out the following steps: (i) compute the disjunctive normal form of W(1), (ii) drop the disjuncts that involve the non-basic induction variables, and (iii) perform generalization.

The disjunctive normal form of the formula $\{x < m \supset i+1 < m \supset i+1 < n\}$ is $\{x \geq m \vee i+1 \geq m \vee i+1 < n\}$. After dropping the disjunct that involves the non-basic induction variable *x*, we have $W(1)=\{i+1 \geq m \vee i+1 < n\}$. After performing generalization on $\{i+1 \geq m \vee i+1 < n\}$, we get $W(1) = \{m \leq n\}$. (It is easy to verify that W(1) is true on entry to the loop.) Since *m* and *n* are not modified in the body of the loop, we have $W(2) = \{m \leq n\}$, which is implied by $W(0) \wedge W(1)$. Hence the loop invariant synthesized is $\{x < m \supset y < n \wedge m \leq n\}$. This loop invariant implies that *y* is less than *n* at line 6.

We believe that standard techniques such as those described by Muchnick [61] should allow us to identify the induction variables and to perform the transformations described above to address this limitation of the induction-iteration method.

7.4 Summary

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

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

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

- Certain compiler optimizations, such as strength reduction and optimizations to address-calculations, *complicate* the task of global verification because they hide relationships that can be used by the induction-iteration method.

- There are several strategies that make the induction-iteration method more effective: First, because certain conditions in a program can pollute L(*j*), instead of using wlp(loop-body, W(*i-1*)) as W(*i*), we compute the disjunctive normal form of wlp(loop-body, W(*i-1*)), and try each of its disjuncts as W(*i*) in turn. Second, we rank the potential candidates according to a simple heuristic, and test each candidate for W(*i*) using a breadth-first strategy, rather than a depth-first one. Finally, forward propagation of information about array bounds (such as range analysis) can substantially reduce the time spent in the induction-iteration method (it reduces the time needed to verify that a W(*i*) is true on entry, and it can eliminate the need to use generalization to synthesize a loop invariant). In our implementation of the safety checker, we use the results of range analysis as assertions to assist the induction-iteration method.

- Verifying an interprocedural version of an untrusted program can take less time than verifying a manually inlined version because the manually inlined version replicates the callee functions and the global conditions in the callee functions. This is a place where our analysis benefits from the procedure abstraction.

- It is more expensive to verify programs that have array out-of-bounds violations than programs that are safe. This is because we have to exhaust the

breath-first search of the induction-iteration method before we can conclude that an array out-of-bounds violation is possible.

- In our work, we perform safety checking without any help from the compiler. However, some information that is easy for the compiler to retain, but hard for our analysis to infer, would make the analysis easier. Such information could include: types and sizes of local arrays, pointers to array inside a structure, and correlations among loop induction variables that were hidden due to compiler optimizations. In particular, information about the correlations among loop induction variables would benefit both range analysis and the induction-iteration method.

Chapter 8

Conclusions and Future Work

In this dissertation, we have presented techniques for statically checking whether it is safe for a piece of untrusted foreign code to be loaded and executed in a trusted host system. Our techniques work on ordinary machine code, and mechanically synthesize (and verify) a safety proof. In this chapter, we examine the limitations of our technique and discuss future research directions.

8.1 Limitations

A major limitation of our techniques is that they can only enforce safety properties that are expressible using typestates and linear constraints. This excludes all liveness properties, some safety properties (e.g., array bounds checks for array references with non-linear subscripting, and checks for floating-point exceptions).

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

Like all static techniques, our technique is incomplete. First, the analysis loses precision when handling array references, because we use a single abstract location to summarize all elements of the array.

Second, the induction-iteration method itself is incomplete even for linear constraints. The induction-iteration method cannot prove the correctness of array accesses in a loop if correctness depends on some data whose values are set before the execution of the loop. One such example is the use of a sentinel at the end of an array to speed up a sequential search [84]. The generalization capabilities of the system may fall short for many problems, even though we care only about memory safety. The induction-iteration method could fail in cases where a loop invariant must be strengthened to the point that we end up verifying a large part of the partial correctness of the algorithm.

Third, the type system is not sophisticated enough for handling certain cases. In particular, with the type system described in this dissertation, the meet of two array types that have the same element type but have different numbers of elements will return an array type of size zero. This causes our analysis to lose precision when checking the kernel-device-driver example.

Finally, our analysis is not able to deal with certain unconventional usages of operations, such as swapping two non-integer values by means of “exclusive or” operations.

8.2 Future Research

Despite these limitations, the method shows promise. Its limitations represent potential research opportunities, and we believe that future research could make the analysis more precise and efficient, and continued engineering could make the technique practical for larger programs. We discuss some potential research directions in the sections that follow.

8.2.1 Improving the Precision of the Safety-Checking Analysis

8.2.1.1 Developing Better Algorithms: An interesting research direction is to develop a better typestate system and algorithms to make the safety-checking analysis more precise. A natural starting point would be to address the limitations that we have identified, including better handling of array references, handling the meet of array types with better precision, and developing better heuristics to strengthen the induction-iteration method for array bounds checks.

8.2.1.2 Employing both Static and Run-Time Checks: In addition to improving the precision of the safety-checking algorithms, we can sharpen our analysis by incorporating run-time checking. Rather than rejecting untrusted code that has conditions that cannot be checked statically, we can generate code to perform run-time checks. This is straightforward with our technique: The annotation phase of our analysis generates the safety conditions; we need only to generate run-time checks for the conditions that fail static checking.

To allow run-time checking, we need to address the recovery problem— that is, what should we do if a fault is detected at run-time. The simplest thing to do is to terminate the offending code. This can ensure that the untrusted code has not violated–and will not–violate any of the default safety conditions and the access policy. Unfortunately, simply terminating the offending code may not be enough if the untrusted code accesses shared data or acquires host resources. In general, more complicated actions need to be taken to ensure the integrity of the host. Traditionally, people avoid the recovery problem by, for example, allowing the untrusted code to interact with the trusted host through a well-defined interface [41, 93], or using a transaction model [75] so that the untrusted code can be aborted if a fault does occur.

The notion of typestate is more general than what we have used in this dissertation. For each type, its states correspond to a state machine. An operation can bring each of its operands from one state to another. In principle, one could label certain states as safe states, and other states as unsafe states. The safety policy could include a post-condition in the form of typestates and linear constraints to specify the invariants that must hold when the untrusted code terminates. Corrective actions might be generated automatically by examining the difference between the faulting state and the desired state. For example, we can use typestate to capture resources that are allocated but have not been released at the time of a fault, and generate the corrective actions to release the resources.

8.2.2 Improving the Scalability of the Safety-Checking Analysis

8.2.2.1 Employing Modular Checking: One way to make the safety-checking analysis more scalable is to perform intraprocedural analysis instead of interprocedural analysis. Recent work on interprocedural pointer analysis has shown that pointer analysis can be performed in a modular fashion [16,17]. 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. It may be possible to use such techniques to create safety pre- and post- conditions automatically.

8.2.2.2 Employing Analyses that are Unsound: Another way to make the technique more scalable is to employ analyses that are more efficient but are unsound. Such examples include treating a program that contains loops as though it were acyclic, or terminating the analysis after a fixed number of iterations before it converges. In principle, one can have a series of techniques that vary both in efficiency and power, and apply the techniques starting from the most efficient, and apply more expensive techniques only when there is a need.

8.2.2.3 Producing Proof-Carrying Code: One argument against our approach is that the safety-checker we have implemented is much more complicated than the proof checker used in the Proof-Carrying Code approach. It should be noted that what we have shown in this thesis is just one way to structure our safety checker. In principle, we could separate our safety-checker into a proof generator and a proof checker. The proof generator would be used by the code producer to synthe-

size and attach safety proofs to the machine code, and the proof checker would be used by the code consumer to check the validity of the proofs.

8.2.3 Extending the Techniques beyond Safety Checking

8.2.3.1 Enforcing Security Policy: The work in this thesis has focused on enforcing safety based on access control, which is a form of *discretionary access control.* Discretionary access control policies do not impose any restriction on the usage of information once it is obtained by the untrusted code [73], hence cannot prevent any disastrous information leaks.

This limitation can be addressed by *mandatory access control* policies, where the accesses to be allowed are determined on the basis of predefined rules. A generalization of the mandatory access policy is represented by the information flow model of Denning [22].

Denning's policy is based on a lattice model, where a flow policy is represented by a partial ordered set <S, →>. S is a set of security classes, and → is a partial order, called the *flow relation*. Every variable is assigned a security class. Flow of information from variable x to variable y is permissible *iff* *security_class*(x)→ *security_class*(y). The function *security_class*(x) returns the security class of variable x [87].

It should be possible to extend the safety policy described in this dissertation, [Region : Category : Access] to include a flow policy of Denning, by adding a fourth component " securityclass", and the partial ordered set <S, →>. Instances of the fourth component belong to the set S. We would then check that no infor-

mation of a higher security class ever flows into a location that is allocated for storing information of a lower security class.

One limitation of Denning's flow certification method is that it requires all variables to have a security class *a priori*. This limitation makes it unsuitable for certifying legacy code, or code where such information is not available (such as machine code). Volpano and Irvine [87] show that limitations of flow certification of Denning can be overcome via a secure flow type system, where flow control becomes type checking. Their focus has been on producing principal types (security requirements) for function abstractions.

Treating information flow certification as type-checking fits perfectly with the typestate-checking analysis we have used to enforce safety. A simple extension to our technique would be to extend the typestate system to include an orthogonal "secureflow type" component, and to extend the safety-checking analysis to perform secure flow type checking. As a consequence, our typestate-checking analysis would be able to enforce both discretionary and mandatory access policies.

8.2.3.2 Reverse Engineering: Because our safety-checking analysis works on unannotated binary and recovers typestate information that could have existed in the source code, we could use the typestate checking analysis to reverse engineer machine code into code in a typed source language such as C. The information that is recovered by our analysis from an unannotated binary can be useful for purposes other than safety checking. For example, the type information recovered could be used by a dynamic optimizer to enable optimizations that are

impossible without type information (e.g., to eliminate virtual function calls in object-oriented programs [3]). Another possible usage of this information is to guide a performance tool to choose better spots in a program to insert instrumentation code, and to gather and present the performance data that are more sensible to the human.

The attractive aspect of these uses of our techniques is that we can forsake the most expensive parts of our analysis, global verification. We may not have to care whether a variable is defined. We can even perform intraprocedural analysis, for example, if the signature of a function is available.

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] ActiveX: http://www.microsoft.com/com/tech/ActiveX.asp (2000).

[3] G. Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. *10th European Conference on Object-Oriented Programming*. Linz, Austria (1996).

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

[5] W. Amme et. al. Data Dependence Analysis of Assembly Code. *International Conference on Parallel Architectures and Compilation Techniques* (PACT'98), Paris (October, 1998).

[6] V. Balasundaram and K. Kennedy. A technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations. *SIGPLAN Conference on Programming Language Design and Implementation*. Portland, OR, USA (June 1989).

[7] 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).

[8] A. Birrell, and B. Nelson. Implementing Remote Procedure Calls. *ACM Transaction on Computer Systems* **2**, 1 (February 1984).

[9] R. Bodik, R. Gupta, and M. L. Soffa. Refining Data Flow Information using Infeasible Paths. *Fifth ACM SIGSOFT Symposium on Foundations of Software Engineering and sixth European Software Engineering Conference*, LNCS 1301 Springer Verlag, Zurith, Switzerland (Sept. 1997).

[10] 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).

[11] R. S. Boyer and Yuan Yu. Automated Proof of Object Code for a Widely Used Microprocessor. *Journal of the ACM* **43**, 1 (January 1996).

[12] D. Callahan and K. Kennedy. Analysis of Interprocedural Side Effects in Parallel Programming Environment. *First International Conference on Super-Computing*. Athens, Greece (1987).

[13] M. Caplain. Finding Invariant Assertations for Proving Programs. International Conference on Reliable Software (April 1975).

[14] 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).

[15] 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).

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

[17] 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).

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

[19] D. L. Clutterbuck and B. A. Carre. The Verification of Low-Level Code. *Software Engineering Journal* **3**, 3 (May 1988).

[20] 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).

[21] 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).

[22] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM* **19**, 5 (May 1976).

[23] 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).

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

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

[26] 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).

[27] D. Evans. Static Detection of Dynamic Memory Errors. *ACM SIGPLAN Conference on Programming Language Design and Implementation* (May 1996).

[28] D. Evans, J. Guttag, J. Horning and Y. M. Tan. LCLint: A Tool for using Specifications to Check Code. *SIGSOFT Symposium on the Foundations of Software Engineering* (December 1994).

[29] S. Fink, K. Knobe, and V. Sarkar. Unified Analysis of Array and Object References in Strongly Typed Languages. *Static Analysis, 7th International Symposium, SAS 2000*. Santa Barbara, CA (June 2000). Published in Jens Oalsberg (Ed.), Lecture Notes in Computer Science, 1824. Springer-Verlag (2000).

[30] C. Flanagan, M. Flatt, S. Krishnamurthi, and S. Weirich. Catching Bugs in the Web of Program Invariants. *ACM SIGPLAN Conference on Programming Language Design and Implementation* (May 1996).

[31] R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposia in Applied Mathematics XIX*. American Mathematical Society (1967).

[32] S. M. German. Automating Proofs of the Absence of common runtime errors. *ACM Symposium on Principles of Programming Languages*. Tucson, Arizona (January 1978).

[33] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. *USENIX Annual Technical Conference (NO 98)*. New Orleans, Louisiana (June 1998).

[34] M. Grand. **Java Language Reference**. O'Reilly & Associates, Inc. (July 1997).

[35] R. Gupta. A Fresh Look at Optimizing Array Bound Checking. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. White Plains, New York (June 1990).

[36] B. T. Hailpern. Verifying Concurrent Processes Using Temporal Logic. Lecture Notes in Computer Science 129. G. Goos and J. Hartmanis (Eds.). Springer-Verlag (1982).

[37] S. P. Harbison. **Modula-3**. Prentice Hall (1992).

[38] W. Harrison. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering **3**, 3* (1977).

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

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

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

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

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

[44] 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).

[45] P. Kolte and M. Wolfe. Elimination of Redundant Array Subscript Range Checks. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. La Jolla, California (June 1995).

[46] 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).

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

[48] V. Markstein, J. Cocke, and P. Markstein. Optimization of Range Checking. *SIGPLAN Symposium on Compiler Construction* (1982).

[49] 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).

[50] J.G. Mitchell, W. Maybury, and R. Sweet. Mesa Language Manual. Technical Report, Xerox Palo Alto Research Center (1979).

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

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

[53] 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).

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

[55] 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).

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

[57] 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).

[58] 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).

[59] 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 (Ed.), Lecture Notes in Computer Science, 1473. Springer-Verlag (1998).

[60] 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).

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

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

[63] 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).

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

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

[66] 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).

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

[68] 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).

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

[70] 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).

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

[72] 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).

[73] P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia. Information Flow Control in Object-Oriented Systems. *IEEE Transaction on Knowledge and Data Engineering* **9**, 4 (July/August 1997).

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

[75] 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).

[76] 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).

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

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

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

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

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

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

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

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

[85] 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).

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

[87] D. Volpano, C. Irvine. Secure Flow Typing. *Computer & Security* **16**, 2 (1997).

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

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

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

[91] 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).

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

[93] 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).

[94] 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).

[95] Z. Xu, T. W. Reps, and B. P. Miller. Typestate Checking of Machine Code. Technical Report, University of Wisconsin–Madison (J uly 2000).