SASI Enforcement of Security Policies: A Retrospective*

Úlfar Erlingsson   Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

April 2, 1999
Revised: July 19, 1999

Abstract

SASI enforces security policies by modifying object code for a target system before that system is executed. The approach has been prototyped for two rather different machine architectures: Intel x86 and Java JVML. Details of these prototypes and some generalizations about the SASI approach are discussed.

1 Introduction

A *reference monitor* observes execution of a *target system* and halts that system whenever it is about to violate some security policy of concern. Security mechanisms found in hardware and system software typically either directly

---

*Appears in *Proceedings New Security Paradigms Workshop 1999*.

*Supported in part by ARPA/RADC grant F30602-96-1-0317, AFOSR grant F49620-94-1-0198, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-99-1-0533, National Science Foundation Grant 9703470, and a grant from Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon.

implement reference monitors or are intended to facilitate the implementation of reference monitors. For example, an operating system might mediate access to files and other abstractions it supports, thereby implementing a reference monitor for policies concerning those objects. As another example, the context switch (trap) caused when a system call instruction is executed forces a transfer of control, thereby facilitating invocation of a reference monitor whenever a system call is executed.

To do its job, a reference monitor must be protected from subversion by the target systems it monitors. Memory protection hardware, which ensures that execution by one program cannot corrupt the instructions or data of another, is commonly used for this purpose. But placing the reference monitor and target systems in separate address spaces has a performance cost and an expressiveness cost. The performance cost results from the overhead due to context switches associated with transferring control to the reference monitor from within the target system. The reference monitor must receive control whenever the target system participates in an event relevant to the security policy being enforced.

The expressiveness cost comes from the means by which target system events cause the reference monitor to be invoked, since this restricts the vocabulary of events that can be involved in security policies. Security policies that govern operating system calls, for example, are feasible because traps accompany systems calls. But, increasingly, security policies are today being defined in terms of application-level abstractions and operations. For instance, a web browser might need to enforce a security policy governing how helper applications use web browser resources. If system calls are not used for such operations, then traps are not available for transferring control to the reference monitor; some other means of invoking the reference monitor would have to be found.

An alternative to placing the reference monitor and target systems in separate address spaces is to modify the target system code, effectively merging the reference monitor in-line. This is the basis for software-fault isolation (SFI), which enforces the security policy that prevents reads, writes, or branches to memory locations outside of certain predefined memory regions associated with a target system [14, 12]. In theory, a reference monitor for any security policy could be merged into a target application—provided the target can be prevented from circumventing the merged code. Two prototype systems that reduce this theory to practice are the subject of this paper.

Our prototypes merge security policy enforcement code into the object code for a target system. We chose to work at the level of object code, in part, to minimize the size of the trusted computing base (TCB).[1] Working at the object code level makes available the rich vocabulary of low-level events—machine language instructions—over which any security policy could presumably be crafted.

One of our prototypes transforms x86 assembly language output from the `gcc` compiler; another prototype transforms Java Virtual Machine Language (JVML). With each, security policies are specified using security automata, a specification notation that has been proved expressive enough to define any security policy that is enforceable using execution monitoring [11].

We proceed as follows. The use of security automata for specifying security policies is discussed in §2. Our general approach to merging enforcement code into a target system is the subject of §3. Two prototype realizations of this approach are then discussed in §4. For each prototype, we discuss how the integrity of the enforcement code is protected and give the results of performance experiments. Next, in §5, we contrast our work with related work. And, §6 critiques our approach, offering the conceptual basis for the second-generation security enforcement toolset that we are now constructing.

2 Security Automata

Informally, a *security automaton* involves a (not necessarily finite) set of states, a (not necessarily finite) input alphabet, and a transition relation.[2] The transition relation defines a next state for the automaton given its current state and an input symbol. It is often convenient to define this transition relation using first-order predicates: such a transition predicate is *true* for a current state q, an input symbol s, and a next state q' iff whenever the security automaton is in state q and input symbol s is read, the automaton state changes to state q'. If no transition from the current automaton state can be made for the next input symbol to be read, then the security automaton rejects its input.

Security automata can be regarded as defining reference monitors. The input alphabet corresponds to the events that the reference monitor would

[1] In particular, by working on object code, high-level language processors are not part of the TCB. However, software that performs object code analysis and object code modification is added to the TCB. That software can be relatively modest is size, as described in §4.

[2] A precise definition of security automata is given in [11]. The summary given in this section should suffice for understanding the current paper.

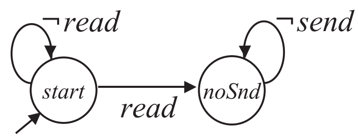

Figure 1: "No messages sent after reading a file".

see. The transition relation encodes a security policy—the automaton rejects sequences of inputs corresponding to target system executions in which the security policy would be violated. For example, Figure 1 depicts a security automaton for a security policy that prohibits message sends after file reads. The automaton's states are represented by the two nodes labeled *start* and *noSnd*. (Automaton state *start* is the initial state of this security automaton.) Predicates *read* and *send* characterize target-system instructions that cause files to be read and messages to be sent, respectively. Thus, the security automaton of Figure 1 rejects any input corresponding to a target system's attempt to execute a send instruction while in state *noSnd* (i.e. after a file read).

3 Merging-in a Security Automaton

Security Automata SFI Implementation (SASI) generalizes SFI to any security policy that is specified as a security automaton. With SFI, new code is added to the target system immediately preceding any instruction that accesses memory (i.e. any read, write, branch, subroutine call, or subroutine return).[3] This new code ensures that (i) all reads and writes to memory will access addresses within the target's data region, (ii) all branches, calls, and returns will transfer control to an instruction within the target program, and (iii) the functionality of these additions cannot be circumvented by the target system.

With SASI, new code is added to the target system immediately preceding every instruction. The added code simulates a security automaton. Specifically, new variables—accessible only to the code added for SASI—represent the current state of the security automaton, and new code—that cannot be circumvented—simulates an automaton state transition. The new code also causes the target system to halt whenever the automaton rejects its input (because the current automaton state does not allow a transition for

[3] In addition, with SFI, the instruction itself is modified in a way that preserves semantics.

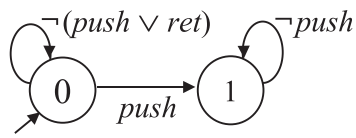

Figure 2: “Push once, and only once, before returning”.

the next target instruction). Thus, the automaton simulation is equivalent to inserting a reference monitor in-line into the target system.

Analysis of a target system often allows simplification of code for simulating a security automaton. Each inserted copy of the automaton simulation is a candidate for simplification based on the context in which that code appears. By using partial evaluation [6] on the transition predicates as well as by using the automaton structure, irrelevant tests and updates to the security automaton state can be removed. Figure 3 depicts the merging of a security automaton specification (given in Figure 2) into a a three-instruction routine that squares a value in `r0`. The security policy in Figure 2 restricts execution to pushing exactly one value onto a stack before returning.

The merge involves four phases:

Insert security automata. Inserts a copy of the security automaton before each target instruction.

Evaluate transitions. Evaluates any transition predicates that can be, given the target instruction that follows each copy of the automaton.

Simplify automata. Deletes transitions labeled by transition predicates that evaluated to *false*.

Compile automata. Translates the remaining security automata into code that, if added at these locations, simulates the operation of the security automaton. `FAIL` is invoked by the added code if the automaton being simulated must reject its input.

Because SASI evaluates and simplifies security automata using only local information, a more-global analysis can sometimes show inserted code to be redundant. For example, in Figure 3, if the `ret` instruction can be reached only through straight-line execution, then `state==0` will necessarily be *false* before `ret` is executed. Thus, the code appearing in Figure 3 before the `ret` instruction might not be needed. The partial evaluator in SASI doesn't attempt global analysis because we feared increasing the size and complexity of the TCB.

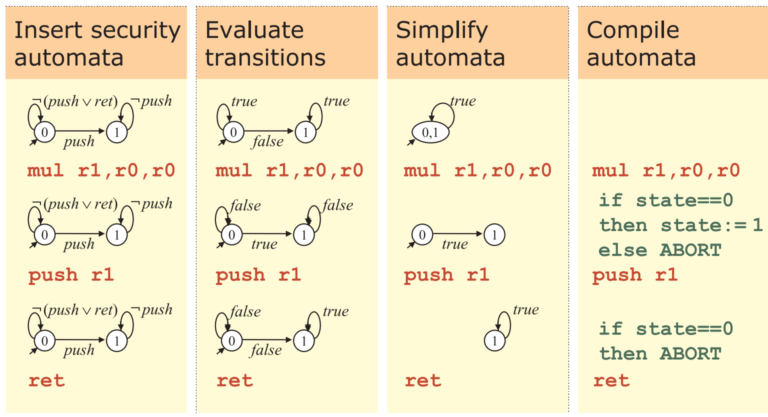

Figure 3: Simplification of inserted code.

4 Two Prototype SASI Implementations

Security policies for our SASI prototypes are represented in SAL (Security Automaton Language). Each SAL specification consists of a (finite) list of states, with each state having a list of transitions to other states. Macros are defined at the start of the SAL specification and are expanded fully bottom-up before use (and therefore may not be recursive). SAL supports only deterministic automata, but this is not a real restriction since non-determinism is easily removed from security automata.

SAL transition predicates are expressions constructed from constants, variables, C-style arithmetic and logical operators, and calls to platform-independent functions and to platform-specific functions:

- For the x86 SASI prototype, the platform-specific functions evaluate to the opcode and operands of the next target instruction, as well as defining the sets of addresses corresponding to data values that can be read/written, and defining addresses of instructions in the target system that can be branch, call, or return destinations.

- For the JVML SASI prototype, the platform-specific functions allow access to class name, method name, method type signature, JVML opcode, instruction operands, as well the JVM state in which the target instruction will be executed.

As an illustration, Figure 4 contains SAL for a JVML SASI specification of the security policy (given in Figure 1) prohibiting message sends after file reads.

```
/* Macros */
MethodCall(name) ::= op=="invokevirtual" && param[1]==name;
FileRead() ::= MethodCall("java/io/FileInputStream/read()I");
Send() ::= MethodCall("java/net/SocketOutputStream/write(I)V");
/*
** The Security Automaton
*/
start ::=
  !FileRead() -> start
  FileRead()  -> hasRead
;
hasRead ::=
  !Send() -> hasRead
;
```

Figure 4: SAL specification for "No messages sent after reading a file".

Associated with each SASI prototype is a *rewriter* which merges a security automaton simulation into object code for a target system. The rewriter operates as outlined in §3. Thus, it inserts code for a security automaton simulation immediately before each target instruction. To construct that code, the platform-specific SAL functions in transition predicates are instantiated with actual values (if they are known) and with code that will compute the values at runtime, otherwise. A generic partial evaluator is next run to simplify the resulting automaton. Finally, object code for the simplified security automaton is generated and inserted in the target code.

The integrity of a reference monitor merged by SASI into the object code of a target system depends on preventing the corruption of that security automaton simulation. This entails

- preventing the target system from modifying variables being used in security automaton transition predicates and variables being used to encode the state of the security automaton,
- preventing the target system from circumventing the code that implements transitions by the security automaton, and
- preventing the target system from modifying its own code or causing other code to be executed (e.g. by using dynamic linking, which most operating systems support), since this could nullify the measures just described for preserving security automaton integrity.

The discharge of these obligations is platform dependent, but there are two general approaches: verification of the object code to establish that the

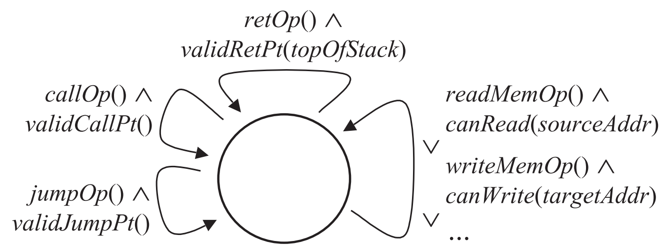

Figure 5: A security automaton for SFI-like memory protection

unwelcome behavior is impossible and modification of the object code to rule out the unwelcome behavior. Both are employed in our prototypes.

4.1 x86 Prototype

Our x86 SASI prototype works with the assembly language output of the GNU `gcc` C compiler. Object code produced by `gcc` observes certain register-usage conventions, is not self-modifying, and is guaranteed to satisfy two assumptions:

- Program behavior is insensitive to adding stutter-steps (e.g. nop's).
- Variables and branch-targets are restricted to the set of labels identified by `gcc` during compilation.

These restrictions considerably simplify the task of preventing a security automaton simulation from being corrupted by the target system. In particular, it suffices to apply x86 SASI with the simple memory-protection policy of Figure 5 in order to obtain target-system object code that cannot subvert a merged-in security automaton simulation.[4] The x86 SASI prototype therefore prepends a SAL description of the security policy in Figure 5 to the SAL input describing any security policy P to be enforced. This effectively merges into the target system a security automaton simulation that cannot be circumvented or corrupted, and where the security automaton simulation enforces security policy P.

Our recursive use of x86 SASI works because the transition predicates in Figure 5 are defined entirely in terms of SAL platform-specific functions

[4]The policy of Figure 5 even deals with the difficulties that arise because valid x86 instructions might exist at non-instruction boundaries in a target system—a bit pattern for one instruction might encompass the bit pattern for another. By limiting branch destination to valid labels, jumping into the middle of an instruction is prevented.

MiSFIT	SASI x86 SFI
`pushl %ebx`	`pushl %ebx`
`leal dirty(,%eax,4), %ebx`	`leal dirty(,%eax,4), %ebx`
`andl offsetMask, %ebx`	`andl segmentMask, %ebx`
`orl writeSegment, %ebx`	`cmpl writeSegment, %ebx`
`movl %edx, (%ebx)`	`jne SASIx86_FAIL`
`popl %ebx`	`popl %ebx`
	`movl %edx, dirty(,%eax,4)`

Figure 6: x86 SASI output for `movl %edx, dirty(,%eax,4)`

which, by virtue of being constructed from information provided by `gcc`, accurately characterize the target program. An informal proof that the transformation suffices proceeds by contradiction, along the following lines. Only branch, call, return, and write instructions can subvert the security automaton simulation. Let i be the first instruction that accomplishes the subversion. Before each branch, call, return, and write instruction, code to check that instruction's operand is added by x86 SASI for the policy in Figure 5. Thus, such checking code must immediately precede instruction i. Since, by assumption, i is the first instruction that accomplishes the subversion, the checking code that precedes it must have been reached and executed. And since the transition predicates are, by construction, accurate, the checking code that precedes i will prevent instruction i from executing. The assumption that i is able to execute and subvert the security automaton simulation is thus contradicted.

Code generated by `gcc` makes indirect stack references by using register `%ebp`, and `%ebp` is guaranteed by `gcc` to point to a valid location in the stack. Our x86 SASI prototype exploits this fact to avoid run-time checks for indirect references through `%ebp` (employing an implementation of Figure 5 that exploits assumptions from `gcc` about `%ebp`). Since indirect references through `%ebp` constitute a significant fraction of the indirect memory references in executables produced by `gcc`, the performance improvement by avoiding these checks can be significant. But exploiting assumptions about code generated by `gcc` in order to reduce the cost of an enforcement mechanism does expand the TCB—a questionable trade-off.

x86 SASI in Action

The memory protection policy given by the security automata of Figure 5 is the same security policy as implemented by MiSFIT, a special-purpose SFI

Benchmark	MiSFIT		SASI x86 SFI	
Page-eviction hotlist	2.378	(0.3%)	3.643	(2.6%)
Logical log-structured disk	1.576	(0.3%)	1.654	(0.5%)
MD5 message digest	1.331	(1.4%)	1.363	(0.1%)

Figure 7: Relative performance of MiSFIT and SASI x86 SFI

transformation tool for Intel x86 operating system extensions [12]. MiSFIT thus constitutes a benchmark against which the performance of x86 SASI can be measured.[5] Consequently, for a set of target systems, we have run x86 SASI with no additional SAL input (so that the policy being enforced is the same as with MiSFIT), and we have run MiSFIT.

The modifications MiSFIT and x86 SASI make to a target system are not very different. Figure 6 shows the Linux assembly output (target operand on the right) generated by MiSFIT and x86 SASI for a `movl` instruction that transfers the contents of register `%edx` into integer array `dirty` at a position specified by the contents of register `%eax`. (In Figure 6, code inserted by x86 SASI is typeset in a slanted font; original target system code is typeset in an upright font.) Notice that MiSFIT actually replaces the original `movl`, whereas x86 SASI only prepends additional instructions. In both, the `%ebx` register is made usable by saving its contents on the stack.[6] For efficiency, x86 SASI uses a load-time generated (i.e. platform-specific) function to determine if the processor flags must be saved for an instruction.[7] This platform-specific function is constructed by using a conservative data-flow analysis within the x86 SASI rewriter.

Figure 7 gives running times for three target systems from [13, 12] that have been processed by MiSFIT and by our x86 SASI prototype. The running times are relative to execution of the unmodified target system, and the numbers shown are averages (with standard deviation in parentheses) over 30 runs on a 266mhz Pentium II running Linux 2.0.30. The "Page-eviction hotlist" benchmark is a memory intensive application; not surprisingly, it has a high overhead in both implementations, because checking code is ex-

[5] In fact, we chose in x86 SASI to make the assumption about `%ebp` only so that we could compare the performance of x86 SASI with MiSFIT, which makes this same assumption.

[6] MiSFIT and x86 SASI thus both require that enough stack space be available for saving the `%ebx` register. This can be ensured at load time. x86 experts will observe that the code from MiSFIT actually contains a subtle bug. If array `dirty` overlaps the stack, then the value of register `%ebx` can be overwritten while it resides on the stack.

[7] The Intel x86 architecture contains such processor flags, implicitly changed by most instructions, that in some rare cases must be saved on the stack like the `%ebx` register.

ecuted for each indirect memory access.

As Figure 7 shows, x86 SASI and MiSFIT produce target systems having comparable performance. But there are target systems where MiSFIT performs considerably better. We do not regard this as discouraging, since MiSFIT is a specialized tool, customized to enforce one specific policy, and MiSFIT optimizes the code it adds. We believe that adding additional analysis and optimization capabilities to the x86 SASI rewriter would improve the relative performance of x86 SASI. More importantly, the flexibility of being able to enforce any policy can make x86 SASI an attractive enforcement tool even if there is some performance cost.

4.2 JVML Prototype

Type-safe languages, such as JVML, provide guarantees about the execution of their programs, including guarantees that imply the simple memory-protection policy given in Figure 5 cannot be violated. A program that does not satisfy this simple memory-protection policy will not be type safe. Our JVML SASI prototype exploits the type safety of JVML programs to prevent a security automaton simulation from being corrupted by the target system in which it resides.

In particular, variables that JVML SASI adds to a JVML object program as part of a security automaton simulation cannot be compromised by the program itself. These variables are inaccessible to that program by virtue of their names and types. Code that JVML SASI adds for the security automaton simulation cannot be circumvented, because JVML type-safety prevents jumps to unlabeled instructions, and the security automaton simulation code segments are constructed so they do not contain labels.[8]

The type-safety of JVML also empowers the JVML SASI user who is formulating a security policy that concerns application abstractions. JVML instructions contain information about classes, objects, methods, threads, and types. This information is made available (though platform-specific functions) in SAL to the author of a security policy. Security policies for JVML SASI thus can define permissible computations in terms of these application abstractions. In contrast, x86 code will contain virtually no information about a C program it represents, so the author of a security policy for x86 SASI may be forced to synthesize application-level events from sequences of object code instructions.

[8]JVML SASI security policies must also rule out indirect ways of compromising the variables or circumventing the code added for policy enforcement. For example, JVML's dynamic class loading and program reflection must be disallowed.

```
...
ldc 1                              ; noSnd state number
putstatic SASIJVML/state           ; change state to noSnd
invokevirtual java/io/FileInputStream/read()I ; read file
...
getstatic SASIJVML/state           ; get current state number
ifeq SUCCEED                       ; if start = state goto SUCCEED
   invokestatic SASIJVML/FAIL()V   ; else violation
SUCCEED:
invokevirtual java/net/SocketOutputStream/write(I)V ; send msg
...
```

Figure 8: SASI JVML enforcement of "no messages sent after reading a file"

JVML SASI in Action

Figure 8 shows code that was produced by our JVML SASI prototype for enforcing the security policy defined in Figure 4 (corresponding to the security automaton of Figure 1 that prohibits sends after file reads). Code inserted by our JVML SASI prototype is typeset in a slanted font; original target system code is typeset in an upright font. Because the target system's instruction to invoke the `FileInputStream/read` method satisfies the security automaton's *read* predicate, this instruction has been prefixed by security automaton simulation code; that code causes an automaton transition to state *noSnd*. And because the target system's instruction to invoke the `write` method satisfies the *send* predicate, it too has been prefixed with security automaton simulation code; that code halts the application if *start* is not the current state of the security automaton.

To gain some understanding about the performance overhead of policy enforcement with JVML SASI, we used our prototype to implement the functionality of Sun's Java 1.1 SecurityManager (SM). Our enforcement is based on a security automaton that specifies checks to be performed at exactly those points in the the Java system libraries where Sun's SM performs runtime security checks. The SAL specification for this security automaton is roughly four pages long and covers java.lang, java.io, and java.util. To ensure the behavioral equivalence of our implementation and Sun's, ours calls the same Java 1.1 `check` functions that SM does. Our implementation is more flexible than Sun's SM—`check` functions as well as new `check` points can be added to ours simply by modifying the security automaton. Modifying the SM in Sun's implementation requires a new release of Java—not just a subclassing of SM—because the places where SM is invoked are fixed by Sun's implementation.

Our JVML SASI implementation of the Java 1.1 SecurityManager turns out to be quite efficient. Microbenchmarks to compare the security overhead in our implementation with that in SM show no statistical difference in the overhead. Moreover, our approach does have the possibility of being significantly cheaper than SM. Sun's SM is invoked at predefined points, whether the check will succeed or not. In settings where access checks are known to succeed—say, because a component is trusted or because of pre-existing access control rights—our JVML SASI prototype does not add any checking code (because the rewriter can simplify the security automaton as it is being inserted). Data for the Blast and the Tar benchmarks in [4] suggest that as much as a fourfold performance improvement can be expected when checking code is eliminated in what arguably are realistic applications.

5 Related Work

SASI is not the first tool to use object code modification as a way of enforcing security policies, nor is it the first work directed at enforcing user-defined security policies.

Object code modification for security policy enforcement and for system monitoring was first used starting in 1969 for an SDS-940 time-sharing system at Berkeley [2]. More-recent work has been directed at enforcing various memory-safety properties that become important for maintaining integrity in extensible systems, where extensions and a base system share a single address space [14, 12]. There has also recently been work directed at enforcing richer classes of security policies for Java programs. Naccio [4] modifies method-call instructions, redirecting them through a wrapper method; Ariel [10] and Grimm and Bershad [5] insert reference monitor code between target system instructions.

With object code modification, the overhead for enforcement can be lowered if object code for the target system is constructed in a way that obviates some of the run-time checks. Overhead also can be lowered if analysis of the object code can identify redundant checks (which are then suppressed). SPIN [1] uses type-safety to avoid run-time enforcement checks, just as x86 SASI exploits the structure of `gcc` code and JVML SASI exploits type safety guarantees provided by JVML. However, the problem with relying on assumptions about the form of the object code is the impact it has on the TCB. Tools that perform object code construction or analysis on which enforcement depends now become part of the TCB; a flaw in such a tool becomes a security vulnerability.

Each of our SASI prototypes depends on a compiler for SAL and on a rewriter (which does some analysis of object code). These software components—written in Java—are part of the TCB for SASI, and they are neither large nor complex. The SAL lexer/parser is roughly 1600 lines, and the partial evaluator adds another 1600 lines. For x86 SASI, the SAL compiler is approximately 850 lines; the rewriter is another 1100 lines, which includes the analyzer necessary for constructing the platform-specific predicates. For JVML SASI, the SAL compiler is approximately 1000 lines; the JVML assembler (3300 lines) and disassembler (650 lines) that complete the picture are components that already exist in the TCB of any JVM implementation.

Note, the compiler that produces JVML (e.g. `javac`) is not part of the TCB for JVML SASI. The required type-safety is established by checking JVML, using a relatively simple analysis. Inexpensive analysis techniques for object code have become an active area of research within the programming languages community because of the potential payoff for reducing the overhead of security policy enforcement. Efficient Code Certification (ECC) provides an analysis method for checking the safety of memory, jump, and stack operations [7]; Typed Assembly Language (TAL) provides a flexible type-safe assembly language that prevents abstractions from being violated [8]; and Proof-Carrying Code (PCC) generalizes from type-safety to allow verification of proofs for even more-expressive classes of properties [9]. These methods could provide a foundation for realizations of SASI for any suitably constructed high-level language compiler and for a variety of platforms.

For supporting user-defined classes of security policies, expressiveness of the policy specification language is crucial. In Ariel [10], Deeds [3], Grimm and Bershad [5], and Naccio [4], security policies are formulated in a language that, at least in part, is processed by a Java compiler. This is no doubt attractive to Java programmers, since security policies can be specified without learning a new language. But by relying on a Java compiler for security policies, that compiler becomes part of the TCB. The ability to use Java for formulating security policies does, however, allow these systems to employ additional state for purposes of policy enforcement, considerably enlarging the set of security policies that can be specified.

Finally, Ariel [10], Grimm and Bershad [5], and Naccio [4], insert checks only at JVML method and constructor calls; Deeds [3] inserts checks only at Java SecurityManager invocations. This restricts the set of enforceable security policies, as only some, not all, instructions in the target system can be halted. For instance, these tools cannot enforce policies that prohibit division by zero, restrict the value of a directly-accessible variable (e.g., a global flag), or implement the stack-access policy of Figure 2. SASI can enforce such policies.

6 SASI in Retrospect

Developing our two SASI prototypes has been instructive, but we do not feel that either x86 SASI or JVML SASI could become a practical tool.[9] First, SAL has proven to be an awkward language for writing real security policies, because SAL forces relevant execution history of target system to be summarized in a single, unstructured "current" state of one or more security automata. Values from the target system that are instances of application-level abstractions must be encoded in order for them to play any role in subsequent enforcement decisions. It is awkward, for example, to write in SAL a policy specifying that the same character string is used in a sequence of operations—SAL cannot directly store strings in automaton states. By changing SAL so that enforcement state can contain typed variables—with types that may be application-level abstractions (e.g., strings)—we believe this difficulty would be eliminated.

A second difficulty with our SASI prototypes concerns what events can be used in defining security policies. A reference monitor that checks every machine language instruction initially seemed like a powerful basis for defining application-specific security policies. In practice, this power has proved difficult to harness. Most x86 object code, for example, does not make explicit the application-level abstractions that are being manipulated by that code. There is no explicit notion of a "function" in x86 assembly language, and "function calls" are found by searching for code sequences resembling the target system's calling convention. The author of a security policy thus finds it necessary to embed a disassembler (or event synthesizer) within each SAL security policy description. This is awkward and error-prone. One solution would be to build a SASI that modified high-level language programs rather than object code. A security automata could be merged into the C++ program (say) for the target system rather than being merged into the object code produced by the C++ compiler. But this is unattractive, because a SASI that modifies C++ programs adds the C++ compiler to the TCB.

The approach taken in JVML SASI seems the more promising way to

[9]However, demos of these tools are available on WWW. Access `http://sasi.cs.cornell.edu/x86demo/` for our prototype x86 SASI processor and access `http://sasi.cs.cornell.edu/javademo/` for our prototype JVML SASI processor.

handle security policies involving application-level abstractions. That is, we advocate relying on annotations of the object code that are easily checked and that expose application-level abstractions. Our current work on building a second-generation of SASI tools is concentrating on JVML for this reason. The approach, however, is not limited to JVML code or even to type-safe high-level languages. Object code for x86 could include the necessary annotations by using the ECC [7] and TAL [8] approaches mentioned above.

Finally, we were pleasantly surprised to observe that our object code modifications do not affect program correctness. High level languages, by distancing the programmer from the assembly language, help prevent program correctness from depending on certain properties of programs—data placement, relative offsets between labels, and so on. This distance gives SASI the latitude to add enforcement code.

Acknowledgments This work has benefited from many lengthy discussions with Greg Morrisett, Robbert van Renesse, and the members of the TACOMA group at Cornell University. Discussions with David Evans helped us sharpen some of our arguments. Dexter Kozen, Andrew Myers, David Walker, Michal Cierniak, and the NSPW program committee provided helpful comments on a draft of this paper.

References

[1] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proc. Symposium on Operating System Principles (SOSP'95)*, pages 267–284. ACM Press, December 1995.

[2] P. Deutsch and C.A. Grant. A flexible measurement tool for software systems. In *Information Processing (Proc. of the IFIP Congress)*, pages 320–326, 1971.

[3] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proc. 5th Conf. on Computer & Communications Security*, May 1998.

[4] D. Evans and A. Twyman. Policy-directed code safety. In *Proc. IEEE Symposium on Security and Privacy*, May 1999.

[5] R. Grimm and B.N. Bershad. Providing policy-neutral and transparent access control in extensible systems. In C.D. Jensen J. Vitek, editor, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 317–338. Springer-Verlag, 1999.

[6] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[7] D. Kozen. Efficient code certification. Technical Report TR98 1661, Cornell University, January 1998.

[8] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language (extended version). In *Proc. Principles of Programming Languages (POPL'98)*, pages 85–97, January 1997.

[9] G. Necula. Proof-carrying code. In *Proc. Principles of Programming Languages (POPL'97)*, pages 106–119, January 1997.

[10] R. Pandey and B. Hashii. Providing fine grained access control for mobile programs through binary editing. Technical Report TR98 08, University of California, Davis, August 1998.

[11] F.B. Schneider. Enforceable security policies. Technical Report TR98 1664, Cornell University, January 1998.

[12] C. Small. A tool for constructing safe extensible C++ systems. In *Proc. 3rd Conference on Object-Oriented Technologies and Systems*, June 1997.

[13] C. Small and M. Seltzer. A comparison of OS extension technologies. In *Proc. 1996 USENIX Technical Conference*, pages 41–54, January 1996.

[14] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault isolation. *Operating System Review*, 27(5), 1993.