Intrusion Detection via Static Analysis

David Wagner
U.C. Berkeley
daw@cs.berkeley.edu

Drew Dean
Xerox PARC
ddean@parc.xerox.com

Abstract

One of the primary challenges in intrusion detection is modelling typical application behavior, so that we can recognize attacks by their atypical effects without raising too many false alarms. We show how static analysis may be used to automatically derive a model of application behavior. The result is a host-based intrusion detection system with three advantages: a high degree of automation, protection against a broad class of attacks based on corrupted code, and the elimination of false alarms. We report on our experience with a prototype implementation of this technique.

1. Introduction

Computer security has undergone a major renaissance in the last five years. Beginning with Sun's introduction of the Java language and its support of mobile code in 1995, programming languages have been a major focus of security research. Many papers have been published applying programming language theory to protection problems [25, 24], especially information flow [17]. Security, however, is a many-faceted topic, and protection and information flow address only a subset of the problems faced in building and deploying secure systems. As attackers and defenders are in an arms race, deploying a system with strictly static but incomplete security measures is doomed to failure: this gives the attacker the last move, and therefore victory.

Formal methods, alone, are insufficient for building and deploying secure systems. Intrusion detection systems have been developed to provide an online auditing capability to alert the defender that something appears to be wrong. Unfortunately, most intrusion detection systems suffer from major problems as described in Section 2. We take a new approach to the problem that eliminates many of these drawbacks.

Our approach constrains the system call trace of a program's execution to be consistent with the program's source code. We assume that the program was written with benign intent. This approach deals with attacks (such as buffer overflows) that cause a program to behave in a manner inconsistent with its author's intent. These are the most prevalent security problems. Of course, some security problems are directly attributable to faulty application logic, such as programs that fail to check authentication information before proceeding, and one limitation of our intrusion detection system is that it does not detect attacks that exploit logic errors. Application logic bugs, however, are dwarfed in practice by buffer overflow problems and other vulnerabilities that allow for execution of arbitrary machine code of the attacker's choice [8, 35], and it is the latter type of vulnerability on which we focus.

The rest of this paper is organized as follows: Section 2 discusses related work, Section 3 discusses our framework, Section 4 discusses the models we use, Section 5 discusses our implementation, Section 6 evaluates our results, Section 7 discusses future work, and Section 8 concludes.

2 Related Work

Early work on intrusion detection was due to Anderson [1] and Denning [9]. Since then, it has become a very active field. Most intrusion detection systems (IDS) are based on one of two methodologies: either they generate a model of a program's or system's behavior from observing its behavior on known inputs (e.g., [14]), or they require the generation of a rule base (e.g., [3]). In both cases, these systems then monitor execution of the deployed program or system and raise an alarm if the execution diverges from the model. The current model-based approaches all share one common problem: a truly robust intrusion detection system must solve a special case of the machine learning problem, a classic AI problem. That is, to prevent false alarms, the IDS must be able to infer, from statistical data, whether the current execution of the system is valid or not. The false alarm rate of present systems is a major problem in practice [2].

Ko *et al.*, and others have proposed a very natural solution to this problem: every program should come with a specification of its intended behavior [21, 19, 22, 29]. This, of course, has been the dream of the formal methods community for 25 years, and is as yet unrealized. We believe it is likely to remain unrealized for some time to come. Although Ko *et al.*'s specification language is simple and admits relatively compact specifications, we believe that the need for manually written specifications will dramatically limit the impact of this work[1]. We philosophically agree with the direction of Ko *et al.*'s work, but we propose to side-step its main drawback by automatically deriving the specification from the program.

3. The framework

We would like to detect the case where an application is penetrated and then exploited to harm other parts of the system. To this end, we define a specification of expected application behavior, and then we monitor the actual behavior to see if it ever deviates from the specification. We describe first how we monitor application behavior, and next we propose techniques for automated specification construction.

To reduce the potentially huge volume of trace data, we consider only the security-relevant behavior of the application of interest. The monitoring strategy should then ensure that a compromised application cannot compromise system integrity[2] while still evading detection. In general, it will always be possible for attackers to evade detection in our system if they do not cause any harm, but if they want to cause harm, they will need to interact with the operating system in a way which risks detection.

In many cases of practical interest, we may safely make the following convenient assumption [15]:

Assumption. *A compromised application cannot cause much harm unless it interacts with the underlying operating system, and those interactions may be readily monitored.*

If—as is typically the case[3]—the only way to interact with the operating system is via system calls, it suffices to monitor just the application's system call trace. Since monitoring system call traces is usually straightforward in practice, the bulk of the challenge will be to derive a specification of the application's expected interaction with the operating system.

We derive our specification of expected application behavior from the application source code, along with a fixed model of the operating system. We model the application as a transition system with some (possibly very large) set of states along with some admissible transitions. If we ever detect a system call trace that is incompatible with this transition system, we may conclude that the most likely explanation is that we are under attack: for instance, the adversary may have introduced malicious code of her own choosing and caused it to be executed, e.g., via a buffer overrun or format string attack. Therefore, to detect intrusions, our basic approach is to look for system call traces that could not have been generated by the underlying transition system.

One subtlety is that the adversary may adapt to our methods. Indeed, we later introduce a new type of attack, the *mimicry attack*, which applies to all intrusion detection systems and in some cases may allow the adversary to fool the intrusion detection system by camouflaging the malicious code so that it behaves much like the application would. We do not have a complete defense against mimicry attacks, but we make some progress towards quantifying resistance against this type of attacker tactic. See Section 6 for details.

Our intrusion detection system does not detect all attacks, but it does allow us to detect one of the most common effects of a penetration: execution of corrupted code. We observe that, in practice, once an attacker has compromised the target application, she will often download some 'exploit code' of her choosing into the application and use it to execute various operations with the application's privileges. Since this exploit code is not originally present in the application source code, if it is ever executed we expect to see behavior that is incompatible with the source code and thus to detect the attack.

One problem is that transition systems derived directly from the source are usually too complex to be useful. We could naively start a second 'slave' copy of the application running on the same inputs in an interpreter that simulates all interactions with the outside world, checking at every step whether we obtain the same system call trace from both the master and the slave. This naive replication strategy could probably be made to work, but it has two important disadvantages. First, replication may be hard to implement, because it is likely to be very difficult in practice to remove every last shred of non-determinism from the application (e.g., random number generators, process scheduling, timing channels, interaction with the outside environment, etc.) [23]. Second, and more importantly, the slave is exposed to the same risks as the master: any set of inputs that tickles a security flaw in the master is likely to trigger the same flaw in the slave as well and thereby escape detection.

[1] However, one promising direction to remedy these limitations can be found in Ko's recent work on blending manual rule bases with automated specification generation [20]. Note that others have used runtime techniques to identify program invariants [12]; however, because the identified invariants concern dataflow, rather than sequencing of system calls, they do not seem to be well-suited to intrusion detection.

[2] We do not consider denial of service attacks in this work.

[3] We do not claim that the assumption is always true. Some operating systems are starting to include partial exceptions to this rule (e.g., user-level networking). However, few security-critical applications use these exceptional features, so we can simply forbid their use: the rare application which uses these features may introduce false alarms, but at least malicious code will not be able to exploit the special features in an attack.

We tackle these problems by simplifying the transition system greatly, abstracting away unnecessary complexity. Since we care only about the sequence of system calls issued, we prune away all other aspects of the model, even to the point of disregarding the contents of local variables, data structures, and all other data flow. We then simulate the simplified transition system in an interpreter with correspondingly minimal operational semantics. This abstraction process has the potential to fix the problems of naive replication: it can be very fast, because most of the code has been pruned away; we can afford to deal with non-determinism, since the transition system has been drastically simplified (for instance, non-deterministic finite automata are not much more expensive to simulate than deterministic finite automata); and the minimal operational semantics may remove many of the pitfalls of C (e.g., buffer overrun attacks will not affect a model that ignores the contents of all buffers).

To summarize our approach: We first pre-compute a model of expected application behavior, built statically from program source code; then, we monitor the program and check its system call trace for compliance to the model at runtime. The primary challenge is in automating model generation, which we discuss next.

4. Models

In this section, we propose a sequence of models that we use to specify expected application behavior: first, a trivial model to illustrate the main idea; then, the callgraph model; third, a refinement, the abstract stack model; and finally, the low-overhead digraph model.

Each model is intended to satisfy a common *soundness* property: false alarms should never occur. To achieve this goal, we must make a number of mild assumptions about our operating environment. We consider only portable C code that has no implementation-defined behavior: for example, we assume that there are no intentional array bounds violations, NULL-pointer dereferences, or other memory errors; we assume there is no function pointer arithmetic or type-casting between function pointers and other pointers; and we assume there is no application-defined runtime code generation. These assumptions are not critical: violations may introduce false alarms but will never cause us to miss attacks we otherwise would have detected. Nonetheless, in our experience the security-critical applications in widespread use do conform to these assumptions.

From a formal language viewpoint, all of our models involve recognizing a sentence in a regular or context-free language. However, this viewpoint is much less intuitive than dealing directly with automata and will not be discussed further. For ease of discussion, we will refer to terminating programs and finite or pushdown automata, as appropriate. All of our results directly extend to non-terminating programs.

4.1. A trivial model

We illustrate these ideas by describing a minimalist example of an intrusion detection system following this framework. Let S be the set of system calls that the application can ever make. The set of allowable system call traces—i.e., our model of expected behavior—will then be exactly the regular language S^*. If, at runtime, we ever observe the application issuing some system call not in S, we prevent the system call from executing, kill the application, and sound the alarm.

This model is easy to derive with automated source analysis tools. Because in practice system calls may be easily recognized in source code, the set S may be inferred easily by simply walking the parse tree and pattern-matching for system call invocations.

Such an approach is simple, easy to implement, sound, and efficient, but it will fail to detect many attacks. No attack that operates using just system calls from S will ever be detected, and in practice we can expect this failure mode to be common if S is too large. Another problem is that the approach is too coarse-grained, since many common system calls are too dangerous to allow without any restrictions. For example, if the `open()` system call is included in S, attackers will be free to modify any file whatsoever at any time without fear of detection. Furthermore, this naive approach scales poorly to large applications, which are exactly the ones at greatest risk for intrusions, because large applications yield large sets S. Consequently, a more precise model is needed.

4.2. The callgraph model

The foremost problem with the naive model described above is that we have thrown away all information about the *ordering* of the possible system calls. In this section we show how to retain some ordering information.

One clean way to represent information on the ordering of possible system calls is to express our model as a regular language over Σ, the set of system calls. For ease of model generation, it is convenient to use an equivalent representation of the model as a non-deterministic finite automaton (NDFA). We describe next how to use a NDFA to characterize the expected system call traces.

Building the model Deriving the model is a simple application of control-flow analysis. We first build the control-flow graph $G = \langle V, E \rangle$ associated with the program source code. We assume that each node of the control-flow graph executes at most one system call and that we can recognize

```
f(int x) {
  x ? getuid() : geteuid();
  x++;
}
g() {
  fd = open("foo", O_RDONLY);
  f(0); close(fd); f(1);
  exit(0);
}
```

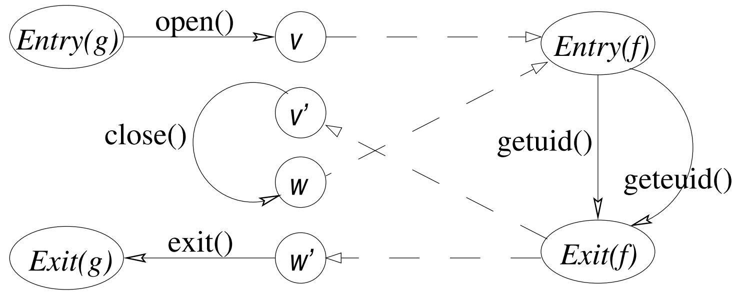

Figure 1. An example C program (left), and its associated callgraph model (right). Transitions to Wrong **are omitted to avoid cluttering the diagram. Dashed lines indicate interprocedural edges, which are represented as ϵ-transitions in the NDFA.**

where system calls occur. Then we note that the control-flow graph can naturally be viewed as a specification of a NDFA with statespace $V \cup \{\mathsf{Wrong}\}$, transitions induced by E, and alphabet Σ. Each edge $v \to w \in E$ of the control-flow graph induces a transition $v \overset{a}{\to} w$ of the automaton, if there is a system call a at node v, or the ϵ-transition $v \overset{\epsilon}{\to} w$ otherwise; ϵ-transitions represent transfer of control where no system call is executed. Every *proper state* (i.e., each state $v \neq \mathsf{Wrong}$) is considered an accepting state. The special state Wrong is non-accepting and contains a self-loop $\mathsf{Wrong} \overset{a}{\to} \mathsf{Wrong}$ on every $a \in \Sigma$; when a node v contains no outgoing transitions on some symbol $a \in \Sigma$, we add an implicit transition $v \overset{a}{\to} \mathsf{Wrong}$. The resulting automaton is non-deterministic because in general we cannot statically predict, for example, which branch of an if-then-else expression will be taken at runtime. See Figure 1 for an example.

We use this automaton as our model of expected behavior, so that an observed trace is accepted only if it is accepted by the NDFA. We call this the *callgraph* model. Note that this model throws away a lot of information about the execution of the application: in particular, we ignore all of its internal state other than the program counter. Nonetheless, it preserves a soundness property:

Claim. *There are no false alarms when using the callgraph model.*

The claim follows from the observation that, by construction, every possible path of execution through the control-flow graph corresponds to an accepting path of the NDFA, and thus every dynamically-possible execution trace will be accepted by the NDFA.

Monitoring algorithm When monitoring the application, we simulate the operation of the NDFA on the observed system call trace, resolving non-determinism by exploring all possible paths in breadth-first order. This requires $O(|V|)$ operations per observed system call. Note that more efficient techniques exist—for instance, the NDFA may be converted to a DFA, either ahead of time or on the fly, and caching may be used to speed up the simulation [18]—but we have not explored any of these alternatives. See Section 5 for more implementation details, and Section 6 for measurements of our implementation's performance and detection power.

Function calls One issue not mentioned so far is how to deal with function calls. After we generate a control-flow graph for each procedure, we connect them together: we split each call site v into two nodes v, v' and add extra edges $v \to \mathrm{Entry}(f)$ and $\mathrm{Exit}(f) \to v'$ for each function f that could be called from v. See the dashed edges in Figure 1 for an example. Here $\mathrm{Entry}(f)$ and $\mathrm{Exit}(f)$ denote the unique entry and exit nodes for f, as might be expected. This so-called *monomorphic* (or context-insensitive) analysis produces a single large graph that may be analyzed as above.

Imprecision in the model One limitation of the callgraph model is that it includes impossible paths, due to the monomorphic treatment of function calls. In particular, consider two call sites v, w that both call the same function f; then the expanded control-flow graph will contain paths of the form $v \to \mathrm{Entry}(f) \to \cdots \to \mathrm{Exit}(f) \to w'$. See Figure 1 for an illustrated example. Such an impossible path cannot occur in any real execution, because function calls will always return to the site where they were called from. Unfortunately, a NDFA is unable to express this constraint, so we end up with impossible paths through the automaton.

Impossible paths in the callgraph model are a problem in

```
f(int x) {
  x ? getuid() : geteuid();
  x++;
}
g() {
  fd = open("foo", O_RDONLY);
  f(0); close(fd); f(1);
  exit(0);
}
```

$$
\begin{array}{rcl}
\mathrm{Entry}(f) & ::= & \texttt{getuid()}\ \mathrm{Exit}(f) \\
 & | & \texttt{geteuid()}\ \mathrm{Exit}(f) \\
\mathrm{Exit}(f) & ::= & \epsilon \\
\mathrm{Entry}(g) & ::= & \texttt{open()}\ v \\
v & ::= & \mathrm{Entry}(f)\ v' \\
v' & ::= & \texttt{close()}\ w \\
w & ::= & \mathrm{Entry}(f)\ w' \\
w' & ::= & \texttt{exit()}\ \mathrm{Exit}(g) \\
\mathrm{Exit}(g) & ::= & \epsilon
\end{array}
$$

while (true)
 case pop() of

$\mathrm{Entry}(f)$	$\Rightarrow$ push($\mathrm{Exit}(f)$); push(`getuid()`)
$\mathrm{Entry}(f)$	$\Rightarrow$ push($\mathrm{Exit}(f)$); push(`geteuid()`)
$\mathrm{Exit}(f)$	$\Rightarrow$ no-op
$\mathrm{Entry}(g)$	$\Rightarrow$ push(v); push(`open()`)
v	$\Rightarrow$ push(v'); push($\mathrm{Entry}(f)$)
v'	$\Rightarrow$ push(w); push(`close()`)
w	$\Rightarrow$ push(w'); push($\mathrm{Entry}(f)$)
w'	$\Rightarrow$ push($\mathrm{Exit}(g)$); push(`exit()`)
$\mathrm{Exit}(g)$	$\Rightarrow$ no-op
$a \in \Sigma$	$\Rightarrow$ read and consume a from the input
otherwise	$\Rightarrow$ enter the error state, Wrong

Figure 2. The example C program again (left), with its associated context-free grammar (middle) and the resulting abstract stack model (right).

practice. This imprecision causes our NDFA to be larger than necessary, and attacks that follow these impossible paths will remain undetected. As a consequence, intrusion detection systems based on the callgraph model may in some cases be more permissive than we would like.

4.3. The abstract stack model

We next introduce the *abstract stack* model, which allows us to characterize more precisely the set of possible system call traces by eliminating impossible paths. The idea is to model not only the program counter but also the state of the call stack. We extend our model so that the set of possible system call traces is allowed to form a context-free language. It is then natural to represent this abstraction of the program as a non-deterministic pushdown automaton (NDPDA), or equivalently, a context-free grammar.

Building the NDPDA The pushdown automaton we construct will provide an intuitive model of program behavior. The state of the automaton will be an abstract summary of the state of the application. In particular, the automaton's stack will form an abstract version of the program call stack: each symbol on the automaton's stack will correspond to a single stack frame in the application's call stack, where everything but the return address has been abstracted away.

The construction is as follows. We assume that we are given a global control-flow graph $G = \langle V, E\rangle$ that includes interprocedural call edges. We generate a NDPDA with stack alphabet $V \cup \Sigma$, input alphabet Σ, and transitions given as follows. Suppose first that there is a node $v \in V$ on the top of the stack. If v is a function call site referencing a procedure f, we pop v off the stack, push the corresponding return site v', and finally push $\mathrm{Entry}(f)$ on to the stack. If v is a function exit node, we pop v. If v is a non-call node, we pop v, push s if v issues the system call $s \in \Sigma$ (otherwise, we do not push anything for nodes that do not make system calls), non-deterministically select some successor w of v with $v \to w \in E$, and finally push w. On the other hand, if $s \in \Sigma$ is at the top of the stack, we attempt to match s against the current input symbol s': if $s = s'$, we consume the current input symbol and pop s off the stack; otherwise, we enter the state Wrong and reject the input string. As in the callgraph model, all proper states are accepting states. See Figure 2 for an example.

This construction of the NDPDA ensures that every sequence of operations to the program call stack during a normal application execution will be among the set of paths explored during the simulation of NDPDA. Since the NDPDA is non-deterministic, other paths may also be explored, but we can be sure that the correct one will not be omitted. At the same time, the increased precision of the abstract stack model makes it less likely that real attacks will go undetected.

The context-free model In our implementation, the NDPDA is constructed directly. However, as the construction is rather detailed, it may be easier to consider building an (almost, as explained below) equivalent context-free grammar for the program, with non-terminals taken from V, terminals in Σ (the set of system calls), and rules given as follows[4]. If v is a function call site with corresponding return site v', we add the rule $v ::= \mathrm{Entry}(f)\ v'$ for each function f that could be called from v. For each non-call node v and each successor w of v, we add the rule $v ::= a\ w$ if there is a system call $a \in \Sigma$ at v, or the rule $v ::= w$ otherwise. Fi-

[4] There are some complications with `setjmp()` and other non-standard forms of control flow; see Section 5.1 for extensions to handle them.

nally, for each function f in the program, we add the ϵ-rule $\mathrm{Exit}(f) ::= \epsilon$. This grammar is quite natural [27, 28, 6, 7].

The simplification referred to above is that the NDPDA, by construction, also accepts all prefixes of sentences generated by the grammar above. The actual grammar would be more complicated if it took this into account.

The NDPDA described earlier may be obtained by applying the trivial top-down construction to the context-free grammar obtained above (similar to $LL(0)$ parsing, except that we keep the conflicts and thus obtain a non-deterministic automaton). This top-down construction is convenient because its operation corresponds closely to execution in procedural languages such as C. See Figure 2 for an example.

Monitoring algorithm To detect attacks, we must monitor the system calls issued by the application and simulate the operation of the NDPDA on those inputs. It turns out that efficient simulation of the NDPDA is a significant theoretical and engineering challenge, especially as we scale up to intrusion detection on very large applications.

The most naive approach is to exhaustively search through all possible non-deterministic choices of the NDPDA. In other words, at each time step, we maintain a list of all possible stack configurations of the NDPDA; when a new system call is observed, for each previously possible configuration we compute the set of new configurations the NDPDA might transition to, and update the list of possible stack configurations. However, in practice this approach is untenable for any but the simplest application, because these lists grow exponentially large in the length of the system call trace (in fact, even infinitely large, in the presence of left-recursion).

Less naively, we might hope that standard parsing algorithms might be applicable here. Of course, we cannot use standard parsers (such as `yacc`) because our NDPDA is non-deterministic. It is easy to see that, for every context-free grammar Γ, there is some program which generates Γ, and in practice, real applications produce grammars with considerable non-determinism and complexity. So, we need an efficient algorithm for online parsing of general context-free languages.

It is also important to have a top-down parsing routine. As described in Section 5, dealing with some of the special features of the Unix runtime environment requires us to occasionally step outside of the context-free framework and perform operations directly on the set of possible stack configurations. Real programs execute in a roughly top-down fashion—we start executing `main()` before executing any of its callees—so this seems to rule out bottom-up parsing. Unfortunately, much of the work in the literature on recognizing general context-free languages (e.g., the CYK, Earley, Tomita, and GLR techniques [37, 10, 16, 33]) uses bottom-up methods.

Consequently, we were forced to develop new techniques for efficient top-down parsing. A full description of our algorithm is outside of the scope of this paper, but we list a few useful properties of the algorithm that make it well suited for our purposes:

- It supports online parsing: as each system call is observed, we can decide whether the resulting partial trace forms the prefix of a sentence in the context-free language, as required for real-time intrusion detection.
- It is relatively efficient: like other general context-free recognizers, its worst-case running time is cubic in the length of the system call trace. This is likely to be too slow for large applications, but is much better than exponential-time solutions. In practice, we encounter cubic-time behavior only occasionally.
- Most importantly, it supports real-time access to the set of possible top-down parse trees. The key data structure is a representation of the set of possible call stacks as a regular language over the alphabet of stack symbols. This lets us modify this data structure directly whenever we need to step outside of the context-free framework.

More details on this algorithm are available elsewhere [34].

4.4. The digraph model

We next introduce a very simple approach which combines some of the advantages of the callgraph model in a simpler formulation. The basic approach, first introduced in previous work on runtime intrusion detection [14], is to consider windows of consecutive system calls.

Our model will thus be a list of the possible k-sequences of consecutive system calls, starting at an arbitrary point during program execution. In our prototype implementation, we consider only the special case $k = 2$ for simplicity. Note that k-sequences of system calls with $k = 2$ are often referred to as *digraphs*, so we call this the digraph model. We consider here both the special case of digraphs and the general case.

Building the model We could derive the set of possible k-sequences from the control-flow graph in a straightforward fashion, but we observe that there is a more precise approach available if we use the context-free language of possible system call traces, $L(\Gamma)$, as introduced in Section 4.3. To determine whether the sequence $s \in \Sigma^k$ can occur in a system call trace during normal application execution, we simply test whether $(\Sigma^* s \Sigma^*) \cap L(\Gamma) \neq \emptyset$, which is effectively computable [18, 27]. Repeating this test for each $s \in \Sigma^k$ gives a general algorithm to build the desired model. Unfortunately, this precomputation has running time $\Theta(k^3 \times |E| \times |\Sigma|^k)$, which is exponential in k. In practice, it is slow enough that we have only experimented with the $k = 2$ case.

Monitoring algorithm Detecting attacks then becomes easy once we have performed the above precomputation to build a list of the allowed k-sequences. We keep a history of the last $k - 1$ system calls, and when we see a new system call, we check whether the resulting k-sequence is allowed. Thus, the runtime monitoring algorithm is extremely efficient for this model; the trade-off is that the digraph model is less precise than the callgraph or abstract stack model, and thus can be expected to miss more attacks.

5. Implementation issues

We sketched above three theoretical frameworks for implementing intrusion detection using static analysis. In practice, though, there are a number of complications that arise when implementing these ideas. We discuss here some of the important implementation challenges and how to handle them.

5.1. Non-standard control flow

Implementations of control-flow analysis, when intended for optimization, often give up in the presence of non-local control flow (such as signals, `setjmp()`, and so on). However, we have found that, in practice, real applications of interest for intrusion detection often use these features. Therefore, we describe how to augment the modelling frameworks described above to incorporate these forms of non-standard control flow.

Function pointers To build the program call-graph in the presence of function pointers, it is crucial to be able to predict the possible targets of every indirect call through a function pointer. Many sophisticated algorithms for pointer analysis are available in the literature [11, 31, 30], but in our implementation we simply assume that every pointer could refer to any function whose address has been taken. Empirically, even this very crude technique seems to suffice for our purposes.

Signals Many operating systems allow applications to register a signal handler to be executed upon reception of a signal. It is straightforward to statically recognize signal handlers: we simply look for system calls of the form `signal(i,fp)`, which binds the handler `fp` to the signal `i` so that when this signal is received, the function referred to by the function pointer `fp` will be called. Consequently, the real challenge is to augment the model to represent these additional possibilities for control flow.

Naively, one might consider adding to the control-flow graph an extra edge from each node to each possible signal handler to represent this additional control flow. This naive solution would work, but it adds an enormous amount of extra non-determinism to the control-flow graph, so our analysis would become less precise: the intrusion detection system would become significantly slower (because we need to follow more possible paths in the control-flow graph) and poorer at recognizing intrusions (because real attacks might mimic unlikely paths through signal handlers and thereby avoid detection). We would prefer to model signals without incurring these costs.

Fortunately, there is a clean solution available. We exploit the presence of a runtime component in our system:

Principle 1. *If you can arrange to receive an extra event whenever some exceptional path (such as invocation of a signal handler) might be taken, you can often improve the precision of the model.*

In this case, we arrange to monitor not only the system calls the application makes but also the signals the application receives[5], and we ensure that all the extra paths in the control-flow graph are *pre-guarded* by an initial signal reception event. In many Unix operating systems, all signal handlers invoke the `sigreturn()` system call after they return, so we also add a *post-guard* to the end of each extra path, too.

It is straightforward to augment the control-flow graph to ensure that every execution of a signal handler will be bracketed by both a pre- and post-guard. These extra paths in the control-flow graph will not be triggered unless the appropriate signal is received, and to save space they may be implicitly represented and only re-generated on demand, so they are effectively invisible except in the cases where they are necessary. These techniques provide a precise, efficient, and simple way to extend any of the models in Section 4 to reflect the semantics of signals.

The `setjmp()` primitive ANSI C provides a form of non-local control flow that is sometimes used to provide a crude form of exception handling or error recovery: the `setjmp()` primitive saves the stack pointer and other registers, and then `longjmp()` may be called by a subroutine to roll the registers, and hence the stack, back to its saved state. In the callgraph model, we may simply add an extra transition from each `longjmp()` to every possible `setjmp()`,

[5]The ability to monitor signals is conveniently already available with most existing mechanisms for process tracing, since it is used by some debuggers.

but this will not work for the abstract stack model because `longjmp()` modifies the call stack.

We do not know of a good static approach to call stack analysis in the presence of `setjmp()`, but fortunately, there is no need to solve this problem statically. Instead, we extend the runtime monitoring agent. Our monitor maintains a running list of all call stacks that were possible when some `setjmp()` call was visited earlier in this execution trace. Each `longjmp()` call can be emulated by adding this accumulated list to the automaton's current set of states. Since sets of states are represented as regular languages in the abstract stack model (see Section 4.3), the union operations may be implemented efficiently.

As a future extension, we might also enforce the constraint that returning from a function activation invalidates any `setjmp()` it may have called. This would allow us to garbage-collect old `setjmp()` states (thereby reducing storage costs by some unknown amount) and to exclude impossible `longjmp()` targets (thereby improving precision and attack detection power). So far, though, we have not found the need. Our experience has been that `setjmp()` is typically used just often enough that it cannot be completely ignored but rarely enough that the burden of the above simulation techniques is minimal.

In any case, our experience with `setjmp()` suggests the following lesson for hybrid static-dynamic systems[6]:

Principle 2. *Some program properties that are difficult to infer statically may become easier to model satisfactorily when the burden is offloaded to a runtime agent, where available.*

5.2. Other modelling challenges

Libraries Our approach requires a model for each library function that might be called. Therefore, we use a modular analysis to build these models. In particular, we modified the `gcc` compiler to output intermediate analysis output files alongside each object file as it compiled, and we modified the linker to combine the intermediate files into a whole-program analysis. A side benefit was that we could analyze existing software packages by simply using the provided Makefiles to compile them.

We found that library code taxed the limits of our tool more thoroughly than most applications, and a disproportionate amount of our effort was spent on the C libraries. For instance, the GNU `stdio` implementation uses function pointers extensively to emulate an object-oriented programming style; with our naive pointer analysis, the inferred models were too imprecise, so we replaced our automated analysis results with hand-crafted models. In contrast, the database library `libdb` also uses function pointers extensively to parametrize database implementations, but in this case we were willing to accept the imprecision. As a third example, the GNU ELF libraries make heavy use of both `setjmp()` and function pointers to implement exception handling, so we resorted to refining the inferred model by hand in some places to improve its precision.

There are many disadvantages to hand-built models: they are time-consuming to construct; they are difficult to get right (and thus unsoundness and false alarms are a risk); and they make it unpleasant to keep up with changes to the code. Ideally, we would have preferred a more precise automatic analysis so that we could avoid these disadvantages, but in practice even our crude techniques were generally sufficient to get the job done without compromising our primary goals in the few cases where manual analysis was necessary.

Dynamic linking Dynamically linked libraries pose another challenge, because they force us to update the model at runtime. In our implementation, we predict in advance the set of libraries which might be linked in and build models for all of them from source. This can introduce false alarms if our prediction becomes out of date (when, e.g., a new version of the library becomes available), which means that everything must be updated whenever the underlying libraries are. This is not a fundamental limitation, and a more satisfying solution would be to build a model at runtime from object code, but we have not explored this direction because it has not been necessary in our experience. In any case, binding applications to libraries statically has substantial security benefits, because it prevents introduction of Trojan horses via dynamic linking attacks.

Threads Threads pose yet another challenge, because the context-switching operation introduces another type of implicit control flow. If it were possible to reliably receive 'thread context-switch' events (see Principle 1), handling threads would be straightforward; this is no problem for kernel threads, but unfortunately, user threads pose a thorny challenge, and we know of no good general solution. A second issue is that threaded code may contain security vulnerabilities due to synchronization bugs that we do not know how to detect. Because of these challenges, and because no security-critical application we examined used threads, our prototype implementation does not support threaded code.

[6] In the digraph model, neither Principle 1 nor 2 is much help, since no help is available from the runtime agent nor is there any convenient way to monitor `setjmp()` and `longjmp()` calls at runtime. Thus, we are forced to use more conservative techniques. Consider temporarily extending the alphabet with the symbols § and $\pounds$ to represent `setjmp()` and `longjmp()` invocations. We infer that digraph $s_1 s_2$ is possible in some program execution only if (1) $s_1 s_2$ is a possible digraph in the original (unextended) language, or (2) both $s_1 \pounds$ is a possible digraph when the language is extended with $\pounds$ and $\S s_2$ is a possible digraph when the language is extended with § and $\pounds$.

5.3. Optimizations

Irrelevant system calls Up to now we have described an intrusion detection system that monitors all system calls the application invokes, and we originally expected this to be optimal. However, we found that ignoring, e.g., the `brk()` system call can greatly improve performance by reducing the size and ambiguity of the model: in many programs, memory allocation can occur just about anywhere, so seeing a `brk()` system call gives us very little contextual information. This may cause us to miss denial-of-service attacks, but those are beyond the scope of this paper.

In some cases, ignoring certain system calls can even improve the precision of the model. It may sound paradoxical that throwing away information can improve precision, but consider the digraph model: excluding very common system calls gives more context. It is useful to be able to enable this optimization on a per-application basis.

System call arguments The most important optimization is based on the observation that we can gain quite a bit of extra information about the application behavior by examining the arguments to each system call. Since we can often statically predict some system call arguments with little effort, we might as well check them at runtime. We recognize lexically constant system call arguments in our prototype and found that even this extremely crude technique provides noticeable improvements to both precision and performance; see the measurements in the next section.

6. Evaluation

In this section we measure the performance of our three approaches (the *abstract stack*, *callgraph*, and *digraph* models) on a number of typical security-critical applications that one might want to monitor for intrusions. For each model, we measure two variants: a basic implementation that ignores system call arguments, and then an improved implementation that checks all system call arguments that can be statically predicted. In each case, we focus on two key metrics: runtime overhead (*performance*), and robustness of detection against targeted attack (*precision*). As will become clear, our results indicate that there is a strong tradeoff between performance and precision.

Performance In Figure 3, we show the runtime overhead incurred by our system when applied to four representative applications with known security vulnerabilities, `finger`, `qpopper`, `procmail`, and `sendmail`. Of these, `finger` is the smallest (at 1K lines of code, excluding comments, blank lines, and libraries), and `sendmail` is the largest (at 32K lines); the other two are in the middle. The height of each bar in Figure 3 indicates the performance overhead of each model, measured in seconds of extra computation per transaction[7].

The figures use shading to show the effect of checking system call arguments. One might expect that checking arguments could improve performance by reducing ambiguity in the model and thus reducing the number of possible paths through the model that we need to explore at runtime. The measurements confirm this hypothesis, showing that—even though we implemented only an extremely crude data-flow analysis—the performance benefits are substantial.

We initially expected that, due to its complexity, the abstract stack model would be consistently slower than the callgraph model. This is partially confirmed by our experiments, but we were surprised to find many exceptions. For instance, in the case of `procmail`, it appears that the improved precision provided by the abstract stack model more than makes up for the complexity of this model. In general, moving to more precise models may reduce the degree of non-determinism and thereby reduce the number of possible paths explored at runtime.

Note that there is a wide variation in running times. The digraph models are consistently extremely fast (the overhead is too small to measure), but the other models are sometimes vastly slower. For `sendmail`, the callgraph and abstract stack models were so slow that we forcibly terminated the experiment after an hour of computation. Since our goal is for real-time intrusion detection, imposing more than a few seconds of latency onto any interactive application is absolutely unacceptable; an hour is clearly several orders of magnitude too much. Consequently, for some applications, only the digraph model is fast enough; for others, the more sophisticated callgraph or abstract stack models are also workable. We conclude that, in all cases, at least one of the approaches provides acceptable performance, but the type of model must be chosen on a per-application basis.

Our prototype implementation has known problems that make its performance sub-optimal. See Section 7.

Mimicry attacks To motivate the need for precise models, we introduce a new class of attacks against intrusion detection systems, the *mimicry attack*. Recall that one of our primary design goals is to detect not only the attacks that are common today, but also to detect the attacks of the future. Furthermore, our model of the application probably cannot be kept secret from attackers. Consequently, our models need to be precise enough that there is no way for an attacker to cause any harm without deviating from the

[7] We use the word transaction to denote a single interactive event, such as delivery of a piece of email. For interactive applications that are not compute-intensive, we believe the main goal is to avoid introducing more than a few seconds of latency per transaction, and so we measure absolute rather than relative overheads. All measurements were performed on a 450 MHz Pentium II running Java, using IBM's JIT for Linux.

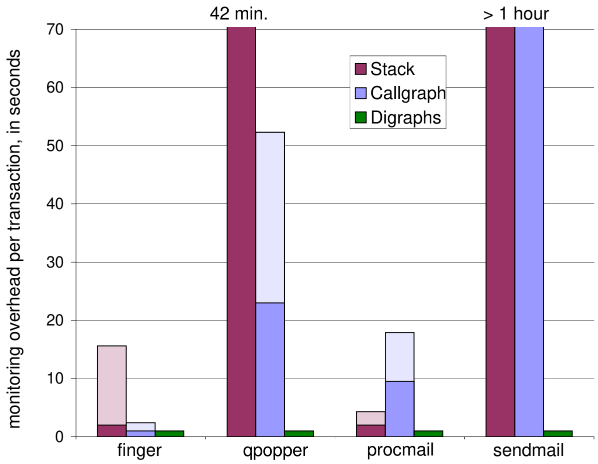

Figure 3. Overhead imposed by the runtime monitor for four representative applications, measured in seconds of extra computation per transaction.

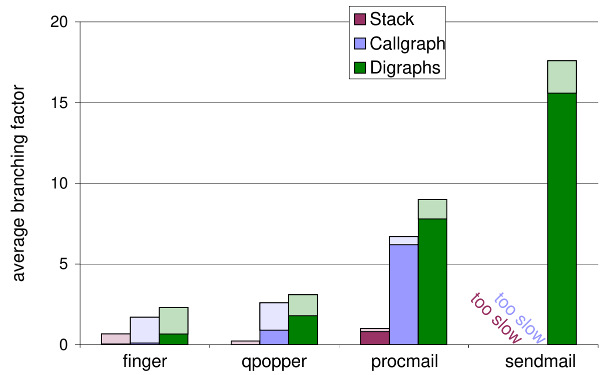

Figure 4. Precision of each of the models, as characterized by the average branching factor (defined later in Section 6). Small numbers represent better precision.

Notes on both figures: For each application, we show measurements for three models using a cluster of three vertical bars: the abstract stack model (leftmost bar), the callgraph model (middle), and the digraph model (rightmost). Each vertical bar uses shading to represent two measurements: the shorter, solid-colored segment represents the case where arguments are checked; the total height of the bar (including both the solid-colored and lightly-shaded regions) shows the case when arguments are ignored.

model, even when the attacker can predict what model we are using. Otherwise, attackers will be free to develop malicious exploit code that mimics the operation of the application, staying within the confines of the model and thereby evading detection by our system despite its harmful effects.

In general, if the attacker somehow obtains control of the application when our intrusion detection automata is in the state s, and if some insecure state s' is reachable from s through any path in the automata, then the attacker will be able to bring the system to an insecure state without risk of detection by synthesizing the system calls that make up the path $s \to \cdots \to s'$. We call this a *mimicry attack*, and we expect that, as intrusion detection becomes more widely deployed, mimicry attacks are unavoidable [26].

Note that imprecise models contain impossible paths, which introduces a vulnerability to mimicry attacks if any of those paths can reach an insecure state. Consequently, the primary defense against mimicry attacks lies in high-precision models.

Precision Unfortunately, we do not know the right way[8] to quantify an intrusion detection system's degree of robustness against mimicry attacks, so we do not have a complete characterization of the precision of our models. Nonetheless, we will attempt to give some intuition for the precision of our models by applying the following metric. Imagine freezing the intrusion detection system in the middle of some application execution trace. There is some set S of system calls that would be allowed to come next without setting off any alarms. We define the *branching factor* to be the size of S. A small branching factor means that the intruder has few choices about what to do next if she wishes to evade detection, and so we can expect that small branching factors leave the intruder most constrained and least able to cause harm. Finally, because we cannot predict at what point during execution the attacker might obtain control of the application, we suggest to measure the *average branching factor* over all normal execution traces. We stress that this metric is insufficient on its own, but it seems to yield a useful first approximation.

Figure 4 shows the precision of our models on our four sample applications, under the average branching factor metric. We can see that checking system call arguments provides substantial precision improvements, because it reduces the number of possible paths through the model, and because some system calls are harmless when their arguments are fixed in advance. For instance, an `open("/etc/motd", O_RDONLY)` call is harmless when

[8] In practice, it may often be difficult even to identify just the set of insecure states of the system.

its arguments are statically known, but otherwise could potentially be exploited by attackers to overwrite arbitrary files on the system. Our experience is that unchecked system call arguments greatly increase our exposure to mimicry attacks. Since checking arguments improves both performance and precision, we conclude that it should always be enabled.

We can also see that, when system call arguments are checked, the abstract stack model is much more precise than the callgraph model, which is itself more precise than the digraph model.

We have also examined the generated models by hand to evaluate how much harm a sophisticated attacker could cause using mimicry techniques. We are confident that all three of the `finger` models leave very little room for attack, due to the fact that the `finger` source code does little else but open a network connection and access world-readable files on the system. Results for the other applications, though, are mixed. The digraph model seems unlikely to resist a mimicry attack, and generally we feel it should not be relied upon for defense against malicious code specially tailored to fool our system. However, the abstract stack model seems to do fairly well: we believe it would successfully limit the harmful effects of any compromise in `qpopper` or `procmail`. On the other hand, for `sendmail`, the generated abstract stack model is too complex for us to make any determination.

We consider it an important open problem to develop a metric or methodology for quantifying the resistance of intrusion detection systems to unforeseen attacks, such as the mimicry attacks introduced above.

Attacks detected We have tested our system on a number of known attacks from the past decade or so. For instance, each of the four applications discussed above has a known security vulnerability; we confirmed that we were able to detect the known attack on those applications.

Probably the most common class of attacks we detect are buffer overruns, which seem to account for perhaps half of all attacks in recent years [8, 35]. Because most existing exploit scripts grab full root privilege and take other distinctive actions (such as launching a shell under the attacker's control) immediately after exploiting the overrun vulnerability, detection is typically straightforward for our tool. Our tool may even be overkill for detecting misbehavior this blatant—many other systems will also detect these attacks, albeit with substantial false alarm rates—but an unusual feature of our tool is that it is also designed to detect some 'stealthy' attacks, as well.

Our approach is also able to detect Trojan horses in trusted software. One current favorite of today's attackers is the `rootkit` toolkit, which replaces some system utilities with a version that contains a backdoor. We verified that our implementation was able to detect when some of these backdoors were exercised (which causes the behavior to deviate from that specified by the original source code).

The most interesting feature of our approach is that it can also detect more exotic attacks, even ones that the designers themselves did not know about. For instance, one extremely subtle attack exploited the ability to pass environment variables to `telnetd` to cause the dynamic linker to link with a shared library provided by the adversary; our system would have detected this attack, and any other dynamic-linking attack that might be discovered in the future, because our model is generated statically with the correct library. More recently, format string attacks have provided another unexpected way to introduce malicious code into vulnerable applications; since our detection mechanism makes no assumptions about how malicious code may be introduced, we can expect our system to apply to format string attacks, as well as to any other ways to take control of vulnerable applications that may be discovered in the future. We feel that these examples illustrate the importance of detecting unforeseen attacks.

Despite these successes, we feel strongly that our tool should not be used as the sole defense against any of these attacks, but instead should be used to complement other techniques. Prevention is often a more effective barrier, and intrusion detection systems are usually best viewed as a backup layer in case the main line of defense is breached.

7. Future work

This work opens up many avenues for future research. The main limitation of our approach is that the run-time overhead is very high for some automata; however, we expect that we could achieve better performance by using more advanced static analysis to get more precise models. Also, the prototype was written in Java; we could recode our system in C or assembly language and directly integrate it into the operating system kernel to reduce the performance overhead substantially. This work also raises the intriguing possibility of reusing the specification that we generate to automatically verify properties of security-critical programs with a model checker. We note that our callgraph model is a finite automaton that appears nearly ideal for a model checker. Our stack model will be more challenging to model check, but there has been theoretical work in this area [5, 13, 32, 36, 4].

8 Conclusions

We have successfully applied static program analysis to intrusion detection. Our system scales to handle real world programs. Also, our approach is automatic: the programmer or system administrator merely needs to run our tools on the program at hand. All other automatic approaches to intrusion detection to date have been based on statistical inference, leading to many false alarms; in contrast, our approach is provably sound — when the alarm goes off, something has definitely gone wrong. Nonetheless, we can immediately detect if a program behaves in an impossible (according to its source) way, thus detecting intrusions that other systems miss.

We relied on a strategic combination of static analysis and dynamic monitoring. This combination yields better results than either method alone and presents a promising new approach to the intrusion detection problem.

Acknowledgements

We thank Alex Aiken, Nikita Borisov, Eric Brewer, Jeff Foster, David Gay, Steve Gribble, Alan Hu, Adrian Perrig, and Dawn Song for useful discussions about this work.

References

[1] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[2] S. Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, 1999.

[3] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Operating system enhancements to prevent the misue of system calls. In *Proc. of the 7th ACM Conference on Computer and Communications Security*, pages 174–183, Athens, Greece.

[4] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *12th Computer Aided Verification*. Springer-Verlag, 2000.

[5] O. Burkart. *Automatic verification of sequential infinite-state processes*, volume 1354 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[6] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*. ACM, 2000.

[7] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*. ACM, 2000.

[8] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proc. 2000 DARPA Information Survivability Conf. and Exp. (DISCEX '00)*, pages 154–163. IEEE Comp. Soc., 1999.

[9] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2), February 1987.

[10] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13:94–102, 1970.

[11] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Lanugage Design and Implementation*, pages 242–256. ACM SIGPLAN, 1994.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions in Software Engineering*, 27(2):1–25, Feb. 2001.

[13] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997. Proceedings of Infinity'97.

[14] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, 1996.

[15] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Sixth USENIX Security Symposium Proceedings*, pages 1–12, San Jose, CA, July 1996.

[16] S. Graham, M. Harrison, and W. Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July 1980.

[17] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM, 1998.

[18] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[19] C. Ko. *Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach*. PhD thesis, U.C. Davis, September 1996.

[20] C. Ko. Logic induction of valid behavior specifications for intrusion detection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 142–153, Oakland, CA, May 2000. IEEE.

[21] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the Tenth Computer Security Applications Conference*, pages 134–144, Orlando, FL, Dec. 1994. IEEE Computer Society Press.

[22] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, Oakland, CA, May 1997. IEEE.

[23] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, Oct. 1973.

[24] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[25] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Jan. 1997.

[26] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, January 1998.

[27] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocecural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*. ACM, 1995.

[28] D. A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*. ACM, 1998.

[29] F. B. Schneider. Enforceable security policies. Technical Report 98-1664, Cornell University, Department of Computer Science, Cornell University, Ithaca, NY, 14853, Jan. 1998.

[30] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Jan. 1997.

[31] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 32–41, St. Petersburg, Florida, Jan. 21–24, 1996. ACM Press.

[32] B. Steffen and O. Burkart. Model checking the full modal mu-calculus for infinite sequential processes. *Theoretical Computer Science (TCS)*, 1999. Special Issue for ICALP'97, to appear August 1999.

[33] M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1–2):31–46, 1987.

[34] D. Wagner. *Static analysis and computer security: New techniques for software assurance*. PhD thesis, University of California at Berkeley, Dec. 2000.

[35] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings 2000 Network and Distributed System Security Symposium*. Internet Society, 2000.

[36] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Computer Aided Verification '98*, pages 88–97. Springer, 1998.

[37] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.