Efficient Intrusion Detection using Automaton Inlining

Rajeev Gopalakrishna Eugene H. Spafford Jan Vitek

Center for Education and Research in Information Assurance and Security
Department of Computer Sciences
Purdue University
{rgk,spaf,jv}@cs.purdue.edu

Abstract

Host-based intrusion detection systems attempt to identify attacks by discovering program behaviors that deviate from expected patterns. While the idea of performing behavior validation on-the-fly and terminating errant tasks as soon as a violation is detected is appealing, existing systems exhibit serious shortcomings in terms of accuracy and/or efficiency. To gain acceptance, a number of technical advances are needed. In this paper we focus on automated, conservative, intrusion detection techniques, i.e. techniques which do not require human intervention and do not suffer from false positives.

We present a static analysis algorithm for constructing a flow- and context-sensitive model of a program that allows for efficient online validation. Context-sensitivity is essential to reduce the number of impossible control-flow paths accepted by the intrusion detection system because such paths provide opportunities for attackers to evade detection. An important consideration for on-the-fly intrusion detection is to reduce the performance overhead caused by monitoring. Compared to the existing approaches, our inlined automaton model (IAM) presents a good tradeoff between accuracy and performance. On a 32K line program, the monitoring overhead is negligible. While the space requirements of a naive IAM implementation can be quite high, compaction techniques can be employed to substantially reduce that footprint.

1. Introduction

The goal of a host-based *intrusion detection system* (IDS) is to identify an attacker's attempts to subvert processes running on the system. An anomaly-based IDS achieves this by identifying program behaviors that deviate from the known normal behavior. Intuitively, IDS algorithms monitor a program by observing *event traces* and comparing those traces to some *expected behavior*. Most approaches use sequences of system calls as a characterization of program behavior. The "normal" program traces can be modeled by observing the program execution on known inputs (*dynamic analysis*) [2, 5, 7, 10, 11, 12, 19], by a domain expert who creates a specification of the program (*manual analysis*) [8], or by automatically creating a specification of the program using static program analysis [1, 3, 4, 16, 17]. All approaches must deal with false positives, when the IDS deems that a legal program event is invalid, as well as false negatives, when an attack goes unnoticed. Clearly false negatives are undesirable as they denote failures of the IDS, but false positives are often more harmful as they hamper correct execution of the program. Dynamic analysis and manual specifications can be accurate as they leverage both domain knowledge and the program's input data, but they are well known to suffer from false positives. Static program analysis techniques can construct conservative program models that are guaranteed not to exhibit false positives. The accuracy of these approaches is illustrated in Figure 1.

The design space of automated techniques for program-model construction must balance the following concerns: *accuracy*, as measured by the number of false negatives; *scalability*, the size of programs that can be handled by the algorithm; and *efficiency*, the runtime overhead of monitoring. There are two aspects of static analysis that affect accuracy in model generation: flow-sensitivity and context-sensitivity. A flow-sensitive model considers the order of execution of statements in the program. The basic model described by Wagner and Dean [17] is an example of a flow-insensitive model where the normal expected behavior is the regular language S^* over the set of program events S (*e.g.* system calls issued from the program text). If the program ever issues a system call outside S, an exception is raised. Such a flow-insensitive approach, while sound and efficient, is highly imprecise in practice because attacks us-

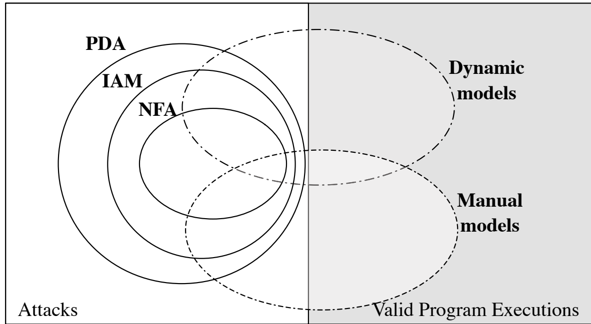


Figure 1. Accuracy of host-based IDS models. The figure shows program traces indicating attack and valid executions. Both Dynamic and Manual models flag some valid traces as erroneous (false positives) and miss some invalid traces (false negatives). Automatically constructed models based on static program analysis are conservative i.e. they do not suffer from false positives, but have varying degrees of accuracy. Pushdown Automata (PDA) are strictly more powerful (i.e. they catch more attacks) than both IAM and Non-deterministic Finite Automata (NFA), although in the absence of recursion, the accuracy of IAMs is the same as that of PDAs.

ing system calls included in S cannot be detected. In large programs, it is quite likely that the set S encompasses all 'dangerous' system calls. For this reason, this paper considers only flow-sensitive models: models that are able to differentiate between sequences of system calls and raise an alert if system calls are issued out of order.

A context-sensitive model keeps track of the calling context of functions and is able to match the return of a function with the call site that invoked it. In a context-insensitive model, event sequences are allowed to start at a call site, go through the called procedure, and return to a different call site. This kind of impossible trace (*i.e.* sequence of events that can not possibly occur in a normal program execution) is a source of inaccuracy for context-insensitive static models. In [17], for instance, a program is represented by a non-deterministic finite automaton (NFA) that is flow-sensitive but does not capture the call-return semantics of high-level programming languages. The advantage of such NFA models is that they impose small monitoring overheads. Context-sensitive models are more accurate at the cost of higher program running times caused by the overhead of maintaining context information.

Context-sensitive models have been investigated by several researchers. In [17], the behavior of a program was captured by a pushdown automaton (PDA), but the authors deemed the runtime costs of the approach prohibitive and argued for simpler models. More recent works [1, 4] have significantly decreased these overheads, yet some monitored programs can still run more than twice as slowly as the original unmonitored code.

While there are obvious reasons why performance overheads are undesirable, there is an additional motivation for keeping this overhead low. Flow- and context-sensitive intrusion detection systems can be tricked into overlooking an attack if the adversary is able to embed the attack in a valid program trace (a so-called mimicry attack [13, 14, 18]). To make such attacks more difficult to carry out, intrusion detection systems must either decrease the granularity of events (*i.e.* observe more of the application's behavior) or be able to perform inferences on the values of arguments to 'dangerous' system calls (*e.g.* discover dynamically that arguments to a call are not valid). These approaches have the potential to improve the accuracy of IDSs but also increase the amount of state needed for verification and thus further increase runtime costs.

In this paper, we present a new abstraction of program behavior referred to as an *Inlined Automaton Model* (IAM) which is as accurate, in the absence of recursion, as a PDA model and as efficient, in terms of runtime overhead, as a NFA. We believe that this abstraction is well suited to be the basis for more expressive intrusion detectors. The contributions of this paper are as follows:

- **Inlined Automaton Model:** The IAM is a flow- and context-sensitive model which is as accurate as a PDA, up to recursion. The paper describes the construction of inlined automata and relates our results to previous work on context-sensitive models.

- **Implementation:** An implementation of IAM is presented. It is based on library interposition. In our system, the events of interest are the invocation of library functions. While it is clearly possible for us to track system calls, we find that library functions give a more accurate model as they are typically more frequent.

- **Empirical evaluation:** The IAM has been evaluated on a benchmark suite that includes two of the same

programs used in [1] to enable direct comparison of results. We have shown runtime performance improvements in both cases. In terms of scalability, our implementation is able to scale to larger programs. We demonstrate scalability by monitoring a 32K line program. Previous experiments with a similar sized program introduced unreasonable overheads and had to be terminated [17].

- **Automata compaction techniques:** We present automata compaction techniques to reduce the space-overhead of IAMs. These techniques are designed to allow users to tune the footprint of the algorithm, with some potential loss of performance.

The remainder of the paper is organized as follows. Section 2 describes existing approaches to statically-constructed model-based anomaly detection. Section 3 describes the construction of IAM and Section 5 discusses different automata compaction techniques. The implementation issues, a solution to reducing non-determinism in IAM, and experimental results are described in Sections 4, 6, and 7 respectively. Section 8 discusses the challenges faced by existing approaches and Section 9 presents the conclusion.

2. Static Analysis-based Automated Intrusion Detection

Static analysis techniques can be used to construct conservative models of program behavior in an automated fashion. The seminal paper on automated program model construction for IDS is by Wagner and Dean [17]. They consider four different models: trivial, digraph, callgraph, and abstract stack. The trivial model represents the expected program behavior using the regular language S^* over the set of system calls S made by a program. It completely ignores the ordering of calls. The digraph model precomputes the possible consecutive *pairs* of system calls from the control flow graph (CFG) of a program and at runtime checks if the pair (*previous system call, current call*) is present in the model. The callgraph model represents all possible sequences of system calls by modeling the expected program behavior using a NFA derived from the CFG of the program. The context-insensitivity arises because only a single instance of a function's CFG is represented in the NFA and this leads to impossible paths (see Figure 2). Finally, the abstract stack model eliminates such impossible paths by modeling the call stack of a program using a PDA. However, [17] demonstrate that in practice the operational costs of a PDA model are prohibitive in both space and time because of having to maintain and search all possible stack configurations on transitions.

Giffin *et al*. [3] evaluate several interesting optimizations to increase precision of NFA models and efficiency of PDAs. The first optimization is to rename system calls (thus extending the set of events S) and allowing the model to distinguish among different invocations of the same function, thus increasing accuracy. The second technique, argument recovery, helps distinguish call sites by recovering static arguments, *i.e.* arguments to functions that can be determined at compile time, for example constant strings or scalar values. Again, this has the effect of enriching the set of observable events and decreasing the number of impossible paths. The last technique proposed in this work consists of a simple, meaning-preserving, program transformation which inserts null calls, *i.e.* calls to a dummy function, at selected points in the program. These calls provide additional context information to disambiguate event sequences. The paper evaluates four null call placement strategies for precision and efficiency. Inserting null calls in functions with a fan-in of five or greater provides a good balance between precision and efficiency. Extending it to functions with fan-in two or greater results in runtime overheads of up to 729%. A PDA model with a bounded runtime stack is also investigated. However, gains in efficiency are observed only by combining this model with null call insertion, which has its own limitations.

The Dyck model [4, 1] improves on the above mentioned null call technique by inserting code around non-recursive call sites to user functions that issue system-calls. The approach basically increases the set of events S accepted by the automaton with unique push/pop symbols; one such guard pair is added for every function call site of interest. This disambiguates call sites to the same target function and thus achieves context-sensitivity. The runtime of the program is affected by the overhead of the instrumentation. The runtime costs can be reduced by dynamic squelching, *i.e.* pruning from the model symbols guarding a function that does not exhibit interesting behavior (*e.g.* does not issue system calls). Nevertheless, slowdowns of 56% and 135% are reported for `cat` and `htzipd`. Recursive calls are not instrumented for performance reasons.

The VPStatic model [1] is a statically-constructed variant of the dynamic context-sensitive VtPath model [2]. It captures the context of a system call by a list (called the virtual stack list) of call site addresses for functions that have not yet returned. This information is obtained at runtime by observing the stack of the monitored process. The virtual stack lists of consecutive system call events are used to determine if that transition is acceptable by the model. While the Dyck model incurs runtime overhead in generating new context-determining symbols, the VPStatic model introduces overhead because of the stack walks necessary to observe existing context-determining symbols. However,

the overhead of stack walks is incurred only at system call events unlike the overhead in the Dyck model which might occur on execution paths without system call events. This difference results in reduced slowdown of 32% and 97% for `cat` and `htzipd` in the case of the VPStatic model. The stack walks make up much of the slowdown.

3. The Inlined Automaton Model

The Inlined Automaton Model (IAM) is a flow- and context-sensitive statically-constructed model of program behavior that is simple, scalable, and efficient. The model is generated by first constructing NFAs for each user function in the program. These automata are constructed by a simple flow-sensitive intra-procedural analysis of the program text. Then, in a second phase, nodes representing call sites are inlined with the models corresponding to the called functions. This process is repeated until all calls have been completely expanded. Recursive calls are treated specially as will be discussed below.

Figure 2 shows an example program and its NFA representation. The NFA abstraction is a union of statement-level CFGs for each function in the program. Each function has unique `entry` and `exit` nodes and call sites are split into `call` and `return` nodes. Call nodes are connected to the entry nodes of the invoked functions and the exit nodes of the invoked functions are connected to the return nodes corresponding to these calls. The context-insensitivity in the NFA model arises because only a single copy of a function's CFG is maintained in the representation. This results in impossible paths being considered by the model. For example, in Figure 2, the system call sequence `(start, write, write, close, end)` is an impossible path. `start` and `end` are special symbols used to denote the start and end of program execution.

Definition 1 *Formally, an ϵ-NFA N for a program P is represented as $N = (Q, \Sigma, \delta, q_0, F)$ [6] where:*

Q is a finite set of states
Σ is a finite set of input symbols
q_0, a member of Q, is the start state
F, a subset of Q, is a set of final states
δ is the transition function that takes a state Q and an input symbol in $\Sigma \cup \{\epsilon\}$ as arguments and returns a subset of Q.

We associate a type T with every state in the NFA representation of a program. So, for each $q \in Q, \exists\, T$ s.t. $T(q) \in \{E, X, C, R\}$, which represent entry, exit, call, and return nodes respectively. We define *successor* of a state q as a set of tuples (s, l), where $s \in Q$ and $\exists\, l \in \Sigma \cup \{\epsilon\}$ s.t. $\delta(q, l) = s$. *Fan-out* of state q is defined as the cardinality of the set *successor*(q). Similarly, we define the *predecessor* of a state q as a set of tuples (s, l), where $s \in Q$ and $\exists\, l \in \Sigma \cup \{\epsilon\}$ s.t. $\delta(s, l) = q$. *Fan-in* of state q is defined as the cardinality of the set *predecessor*(q).

The IAM representation of the program in Figure 3(a) is obtained from the NFA model by inlining all the function calls in the program. The resulting model is context-sensitive because the call-return semantics of function calls is modeled by including a copy of a function's CFG at every call to that function. This model does not have, up to recursion, the impossible paths resulting from context-insensitivity.

Formally, an ϵ-NFA N for a program P given by $N = (Q, \Sigma, \delta, q_0, F)$ is transformed into an ϵ-IAM M given by $M = (Q', \Sigma, \delta', q_0, F')$ where an additional property holds.

Definition 2 *An ϵ-IAM M is an ϵ-NFA where for each $q \in Q'$, if $T(q) = E$ then fan-in$(q) = 1$ or else if $T(q) = X$ then fan-out$(q) = 1$, provided E and X are entry and exit nodes of a non-recursive and non-*`main` *function.*

The final IAM representation shown in Figure 3(b) includes only system call nodes and transitions, and discards the other nodes. This ϵ-free IAM is obtained by performing ϵ-reduction on ϵ-IAM. The definitions of successor and predecessor are the same for an ϵ-free IAM except that ϵ is not an input symbol.

A drawback of inlining is that it may result in state explosion. This indeed is the reason [3] decided not to pursue this approach. The state space can be somewhat limited by restricting the model to states that characterize the observable behavior of the program, *e.g.* system calls, or in our current implementation, calls to library functions. Section 5 discusses space compaction techniques.

Recursion is one obvious limitation of inlining. To ensure termination, it is necessary to treat recursion specially. We perform inlining depth-first. On detecting recursion, we terminate inlining. We connect the call node of the repeating function to the entry node of its previously inlined instance and the exit node of that instance to the current return node. These transitions model the winding phase of recursion. We also connect the call and return nodes of the repeating function to model the unwinding phase of recursion. Examples of recursion bounding for both direct and indirect recursion appear in Figures 4 and 5. Recursion introduces impossible paths, for example in Figure 4, the sequence `(start, open, write, end)` is an impossible path, as it lacks a call to `close` in the unwinding phase, but the path is allowed by the model.

We can relate our approach to the formalization of [1].

```
main( int argc, char** argv) {
   int fd;
   if ( argc == 1 ) {
     write(1, "StdOut", 6);
     foo(1);
   } else {
     fd = open(argv[1], O_WRONLY);
     foo(fd); close(fd);
}  }
void foo(int x) {
   write(x,"Hello World",11);
}
```

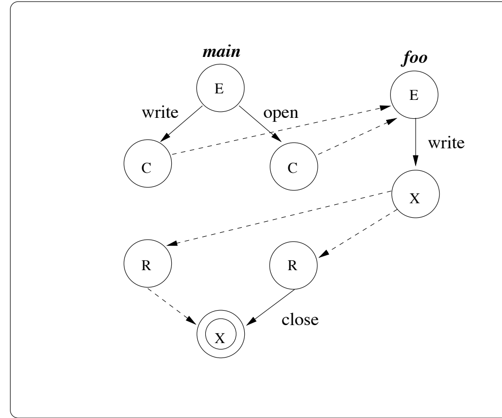
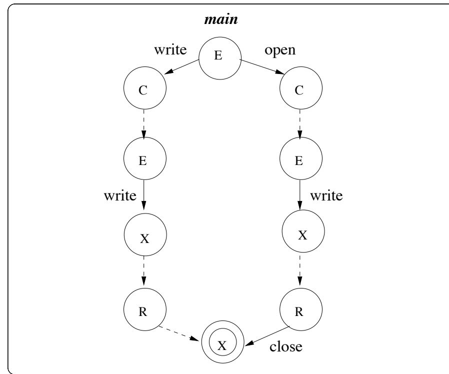
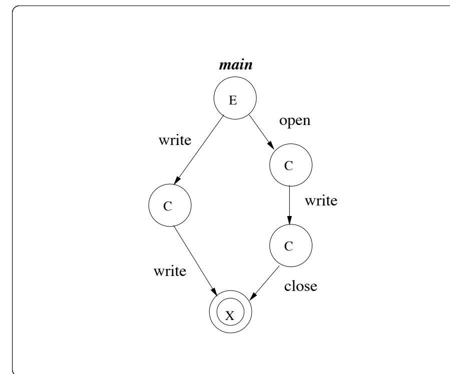


Figure 2. A sample program. In the NFA representation of the program, E, X, C, and R represent entry, exit, call, and return nodes respectively. The dotted lines represent ϵ-transitions in the NFA.



(a)



(b)

Figure 3. (a) The ϵ-IAM representation of the program in Figure 2. (b) An ϵ-free IAM representation.

Theorem 1 *Let $L(IAM(P))$ denote the language accepted by an inlined automaton for some program P, and $L(PDA(P))$ be the language accepted by the pushdown automaton of [1], then we have $L(PDA(P)) \subseteq L(IAM(P))$.*

In the case of recursion-free programs, the languages are equivalent.

4. Monitoring Programs with IAM

Our current implementation of IAM monitors library function calls. The runtime monitor is implemented as a library interposition mechanism [9]. It intercepts calls to library functions and checks them against the model. Figure 6 gives pseudocode for the monitoring algorithm and the data structures used. The algorithm maintains a vector of current states and for every transition, computes the states reachable from that vector. If the set is ever empty, an alert is raised.

4.1. Monitored Events

It should be noted that the algorithm monitors more events than other approaches because we track library functions irrespective of whether they make system calls or not. In the IAMs of the four test programs (see Table 1), only about 25%-50% (26% in `gnatsd`) of the library functions made system calls. So, this generally results in more states and more transitions in our automaton. This bigger size in-

```
main( int argc, char** argv) {
   if (argc > 1) foo(--argc, argv);
}
void foo(int argc, char **file) {
   int fd;
   if ( argc != 0 ) {
     fd = open(file[argc], O_WRONLY);
     write(fd,"Hello World",11);
     foo(--argc, file); close(fd);
  }
}
```

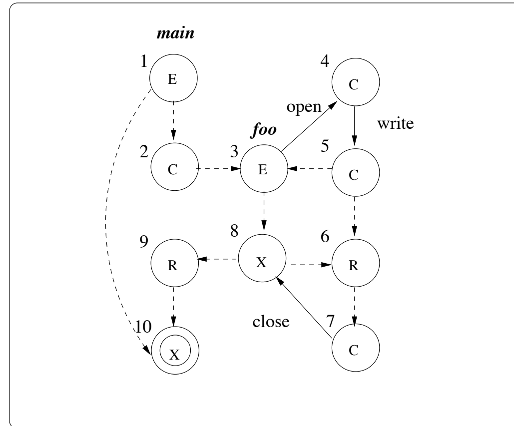


Figure 4. A recursive program. In the IAM representation of the program, dotted lines representing ϵ-transitions have been retained for clarity. The node sequence 1-2-3-4-5-3-8-9-10 which translates to the system call sequence `(start, open, write, end)` is an impossible path.

```
main( int argc, char** argv) {
   if (argc > 1) foo1(argc, argv);
}
void foo1(int argc, char **file) {
   if ( argc != 0) foo2(--argc, file);
}

void foo2(int argc, char **file) {
   int fd; fd = open(file[argc], O_WRONLY);
   write(fd,"Hello World",11);
   foo1(argc, file); close(fd);
}
```

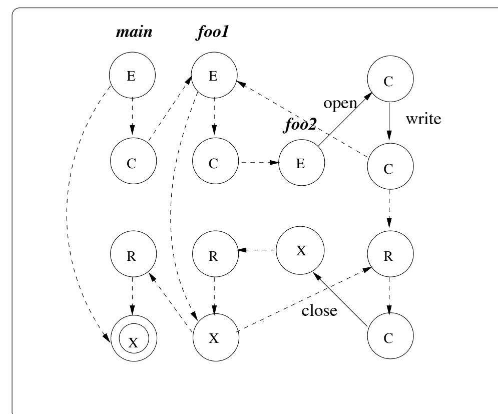


Figure 5. An indirect-recursion program. The program does exactly the same thing as the program in Figure 4 but using mutually recursive functions foo1() and foo2(). The dotted lines representing ϵ-transitions have been retained for clarity.

creases the runtime overhead because of the greater search space. Therefore, modeling libraries instead of system calls is a worst case scenario with the possible exception of a program mostly made up of calls to library functions that make several system calls (e.g. some of the `socket` library functions). In this exceptional case, a model based on system calls would be bigger and slower than our current model. Otherwise, in most cases, the time and space measurements presented in Section 7 can be considered an upper bound for an implementation of a similar approach based on system calls.

4.2. Handling Non-standard Control Flow

Function pointers, `setjmp`/`longjmp` primitives, and signals have to be handled to obtain a sound model. Function pointers in C can be used to make indirect function calls. The functions that can be invoked from a function pointer call site are determined by an analysis of the program that computes the possible values of the function pointer at that program point. But the pointer analysis required to determine this itself requires interprocedural control-flow information. This chicken-and-egg problem can be solved by either ignoring function pointers completely or by combining the construction of control-flow graph with pointer analy-

Data Structures

```
node {
   unsigned int nodeid :  19
   unsigned int funid :  8
   unsigned int succ :  1
}

int funid

node[] curr

node[][] model
```

```
input:  funid, curr, model
output:  curr

succidx ← 0
foreach node n in curr do
   pos ← 0
   repeat
      nd ← model[n.nodeid][pos++]
      if nd.funid = funid then
         if nd ∉ succ then
            succ[succidx++].nodeid ← nd.nodeid
      if nd.succ = 0 then break
   end
end
if succidx = 0 then raise alert
copy succ to curr
```

Figure 6. IAM monitoring algorithm.

sis. Ignoring function pointers is unsound. In the current implementation, we resolve function pointers to all defined functions with the same number and type of arguments as the function pointer invocations. Although this has been sufficient to model our benchmark suite and workloads, this is unsound in the presence of function pointer targets with variable number of arguments and typecasts. Future work should incorporate pointer analysis to more accurately resolve function pointers. This would significantly decrease the model size for programs with heavy function pointer usage especially in the case of IAM which uses inlining.

A call to `setjmp` saves the stack state in a buffer specified by the *env* argument. A call to `longjmp` restores the environment saved by the last call of setjmp with the corresponding *env* argument. In the absence of data flow analysis to determine the pair of setjmp/longjmp calls with the same *env* buffer (lexical matching would ignore effects of aliasing), we connect a longjmp call to every setjmp call in the control flow graph.

Signals are used extensively in privileged programs and network daemons. We identify the signal handlers in a program and construct separate context-sensitive models for them.

4.3. Data Structures

The current implementation of the automaton is based on an ϵ-free IAM model. The automaton is represented by a table of nodes (see Figure 6). Each row in the table corresponds to a state q, and each entry, a node, in the row corresponds to an element of $successor(q)$. Nodes are represented by a node identifier `nodeid` (used as an offset in the table), library function identifier `funid`, and a `succ` bit to indicate if this is the last node (the majority of rows are short; so a bit per node is more efficient than a leading integer). Thus a node represents a tuple (s, l) indicating the transition state s for the input symbol l. The entire structure is packed into 28 bits to conserve space. The 19 bits and 8 bits bit-fields used for the nodes are sufficient to represent all programs we have encountered to date.

The calculations for the automaton compacted by including ϵ-transitions and by using delta successor states (c.f. Section 5) are based on the following data structure[1]. Nodes have three fields: a one byte *funid*, another one byte *length* for the number of bits used in the *offset* field for the delta successors of that state, and finally a pointer to the delta successors. If a node represents a library call then the *funid* is the identifier of the function. If not, it is zero. The value of the *length* field depends on the delta successor with the maximum (absolute) offset from the current state. We cannot use different number of bits for each delta successor depending on its offset because the delta successors of a state have to be of the same size to allow traversal by the runtime monitoring algorithm. Each delta successor contains the *offset* field, one bit to indicate if it is a positive or negative offset, and another bit to indicate if there are more delta successors.

5. Inlined Automata Compaction

The Inlined Model trades off space for time. This trade-off is essential given the performance characteristics of existing approaches to context-sensitive real-time intrusion detection. While IAMs obtain run-time performance better

[1] Note that this data structure is different from the one in Figure 6 which is used in the current implementation.

Program	Software Version	LOC	Description
cat	Solaris 8	<1K	A utility to concatenate and display files
htzipd	LiteZipper-0.1.6	≈7K	A proprietary HTTP server implementation
lhttpd	lhttpd-0.1	<1K	A fast and efficient HTTP server capable of handling thousands of simultaneous connections
gnatsd	gnats-4.0	≈32K	The server daemon component of GNU GNATS, which is a set of tools for tracking bugs reported by users

Table 1. Test programs.

	ϵ-IAM		ϵ-free IAM	
Program	states	transitions	states	transitions
cat	99	342	90	791
htzipd	11,274	15,563	2,821	31,047
lhttpd	650	886	429	1,098
gnatsd	1,286,503	1,969,732	338,736	7,915,678

Table 2. Characteristics of IAM models.

than NFAs (which suffer from performance degradation because of increased non-determinism as a result of context-insensitivity), the footprint of a naive IAM can be rather large. In the previous section we presented a compact data layout for the model. Here we study automata compaction techniques. Table 1 describes the test programs in our benchmark suite and Table 2 gives some basic characteristics of their IAM models.

Coalescing Single-successor States. Straight-line code leads to states with single successors (*i.e.* fan-out of 1). Recall that state transitions occur on calls to library functions. The first compaction technique that we present consists of coalescing a single-successor state with that successor; in effect, having multiple symbols on a transition edge. Formally, For each state $s_j \in IAM$, if $successor(s_j) = \{(s_k, l)\}$, then for each $(s_i, l') \in predecessor(s_j)$, transform (s_j, l') to $(s_k, l'l)$ in $successor(s_i)$. In terms of the transition function, replace $\delta(s_i, l') = s_j$ and $\delta(s_j, l) = s_k$ with $\delta(s_i, l'l) = s_k$. The cost of this optimization is that the monitoring code must keep extra state, a pointer in the array of symbols for the current transition. This cost is rather modest. The space savings come from the fact that a transition can be encoded as a sequence of bytes (one per symbol) and that the state space is reduced as the nodes for the single-successor state are not needed. Every coalesced state would reduce the overall space requirements by approximately 52 bits by removing one row in the model and part of a node.

Figure 7 gives the distribution of fan-out values for states in our benchmark suite. Single-successor states range from 30% in `htzipd` to 58% in `lhttpd`. This shows that there is potential for significant reduction in the number of automaton states using this technique.

Merging Final States. Functions such as `exit`, `_exit`, and `abort` terminate a process. Calls to such functions denote final states in the automaton. We do not need to maintain multiple instances of such final states in the inlined model and can instead have a single final state. This is useful especially when there is extensive use of error-handling routines such as those present in network daemons. The number of zero-successor states (fan-out = 0) ranges from 0.68% in `htzipd` to about 10% in `gnatsd` as shown in Figure 7. For `gnatsd`, which has more than 300,000 states, a 10% reduction is significant. Each final state that can be removed saves approximately four bytes in the overall representation. Furthermore, smaller state space could allow us to reduce the number of bits required for the `nodeid` field (currently 19).

Combining Equivalent Transition Symbols. The above two techniques reduce the number of states in the automaton. This technique takes advantage of the commonality of transition symbols. We know that non-determinism can result in a state having multiple successors. If there are multiple successors for the same transition symbol then one can reduce the overhead by maintaining a single instance of the transition symbol for all those successor states. Formally, the representation for successor states of s_i can be transformed from $\{(s_{j_1}, l), (s_{j_2}, l), \ldots, (s_{j_n}, l), \ldots\}$ to $\{(\{s_{j_1}, s_{j_2}, \ldots, s_{j_n}\}, l), \ldots\}$. In our representation we would save one byte per transition with a common function symbol. Figure 9 shows the average number of successors

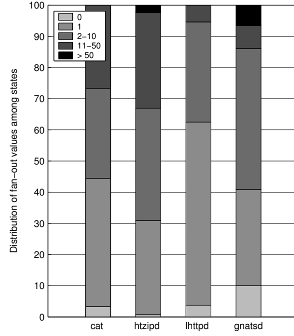


Figure 7. Percentage distribution of fan-out values among automaton states after ϵ-reduction.

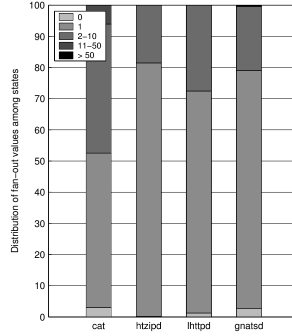


Figure 8. Percentage distribution of fan-out values among automaton states before ϵ-reduction.

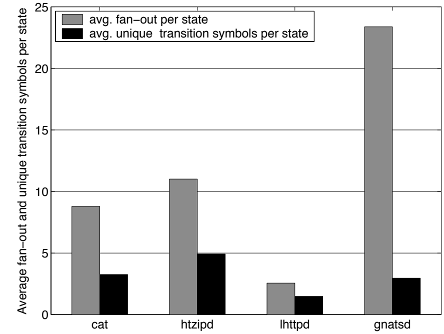


Figure 9. Average fan-out and average unique transition symbols per automaton state.

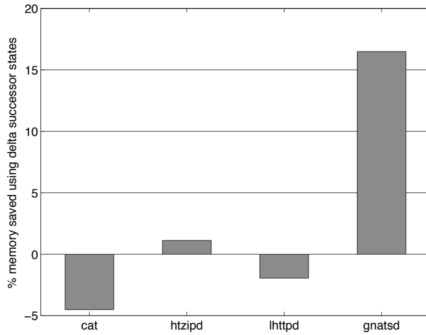


Figure 10. Percentage savings in memory if the model uses delta successor automaton states.

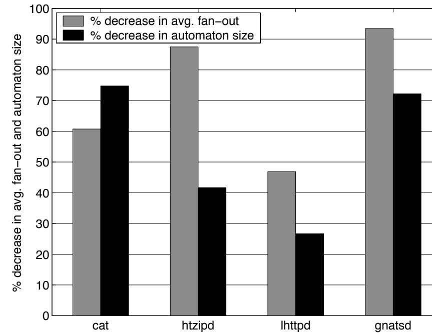


Figure 11. Percentage decrease in fan-out and automaton size if the model includes ϵ-transitions.

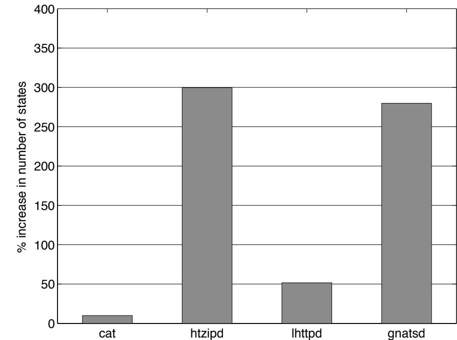


Figure 12. Percentage increase in number of states if the model includes ϵ-transitions.

and unique transition symbols per state. The number of successors being much greater than the number of unique transition symbols per state (especially for `gnatsd`) illustrates the potential for significant space savings.

Maintaining Delta Successor States. States are distinguished by identifiers that are also offsets in the table representing the model. In our example, the default size is 19 bits. Delta successors do not use identifiers for the successors of a state, instead offsets (in the `model`) of the successors relative to the current state. Such successor states are *delta successor states*. Figure 10 shows the memory savings obtained by maintaining delta successor states in the automaton. For `cat` and `lhttpd`, which are smaller programs compared to the other two, the extra information that is needed to use delta successor states outweighs the memory savings and results in a net increase in memory needs. For `gnatsd` however, we can obtain up to 16.49% decrease in memory needs on using delta successor states.

Including ϵ-transitions. Initially an IAM has entry, exit, call, and return nodes for every function instance and call site. Calls to library functions are represented using a single call node because we do not analyze them. Of all these nodes, only the call nodes to library functions are of interest for intrusion detection. Once the basic IAM is constructed, our implementation performs ϵ-reduction. We hypothesized that this would not only yield a more compact representation but also better runtime performance by reducing the search space to only library calls. The final IAM thus includes a single entry node (that of `main`) and call nodes for each library function. However, the compaction increases the degree of non-determinism in the model by introducing more transitions per state. Each automaton state now has a

greater fan-out than before. The maximum fan-outs in the ϵ-free IAMs for `cat`, `htzipd`, `lhttpd`, and `gnatsd` are 35, 136, 33, and 10,028 respectively. In `gnatsd` particularly, there is a high degree of non-determinism in the program text itself. There are `switch` statements with about 80-115 cases in the lexical analyzer and parser subsystems.

Figure 12 shows the percentage increase in the number of states if ϵ-transitions are included in the automaton. The number of states almost quadruples in the case of `htzipd` and `gnatsd`. But the average fan-out per state decreases considerably for all programs (see Figure 11). In the case of `gnatsd`, it drops from 23.3 to 1.5 (a 93.45% decrease). The percentage distribution of fan-out values in this case is shown in Figure 8. Including ϵ-transitions in the automaton and using delta successor states reduces the automaton sizes of these programs by as much as 74.8% for `cat` and 72.2% for `gnatsd` (see Figure 11). Given that single-successor states make up between 49.5% to 81.3% of all states in this case (Figure 8), coalescing can lead to further compaction. However, inclusion of ϵ-transitions may result in additional runtime overhead caused by traversal of the additional states (because, in effect, the monitoring algorithm would have to do an ϵ-reduction at runtime). Our current model representation (the ϵ-free IAM) is highly optimized for time. We believe that in memory-constrained contexts, one can benefit from the above optimizations but expect additional runtime overhead.

Hybrid Automata. Following [1], we discuss a hybrid model that addresses two problems of IAMs: footprint and accuracy. The idea of a hybrid model is to give users more control over space/time tradeoffs. Although there is not necessarily a direct correlation between program size and IAM size (the size of the IAM is affected by issues such as the number of library functions, the number of call sites, the average fan-out), program size remains a good approximation. Very large programs as well as pathological cases will occur and it is desirable to have a strategy to deal with those.

A hybrid automaton model hIAM_k combines inlining with the guarded calls proposed for the Dyck model. A hIAM_k is constructed bottom up starting with leaf functions, i.e. functions that do not call other user functions. Inlining is applied iteratively so that at each iteration, all functions that only call leaf functions are selected for inlining. Modulo recursion, this process will terminate when all functions have been transformed to leaf functions. The main difference with an IAM is that in hIAM_k, the inlining is controlled by a user defined constant k which determines the maximum size for an inlined function. Any function with a number of states larger than k will not be inlined. To retain accuracy, the hybrid model must transform the source program to add guards to calls of non-inlined functions, in the same manner as the Dyck model. Thus, assuming that the program contains a call to `f()` (and that this is the 23rd call site to that function) and the NFA for `f` is larger than k, the following program transformation is applied:

```
f();        ⟹        pre("f",23);
                     f();
                     post("f",23);
```

Clearly, k has an impact on performance: at the extreme if k is 1 then hIAM would degenerate into an instance of the Dyck model.

6. Deterministic Markers

A high degree of non-determinism results in large fan-outs for the automaton states. A larger fan-out translates to greater runtime overhead for the monitoring algorithm which has to check every successor state for matching transition symbols. Furthermore, although the monitoring algorithm starts with a single current state (entry of `main`), non-determinism and the existence of several successor states for the same transition symbol quickly introduce ambiguity about the current state. This causes the monitoring algorithm to maintain a set of current states and check successors for each of them at runtime. The overhead introduced because of this can be significant for a program like `gnatsd` which has a high average fan-out and a low average unique transition symbol per state (see Figure 9). We introduce the concept of *deterministic markers* as a solution to reduce such overhead.

Deterministic markers are unique transition symbols introduced in the program text to reduce the search space (current states and successors) of the runtime monitoring algorithm. Conceptually, they are similar to the renaming and null call insertion techniques described in [3, 4, 1]. The difference is that they are not needed for determining the calling context (inlining takes care of that) but for disambiguating the current state (program counter) and reducing the fan-out of frequently occurring high fan-out states (such as a library function call in a loop followed by severe non-determinism).

Currently, we use such markers only for `gnatsd` along the paths exercised by the workloads. Eleven sites were manually identified and null library calls were introduced. The performance gains were substantial for the minimal effort involved in identifying instrumentation sites. It is reasonable to assume that the selection of instrumentation sites can be automated at model construction time. This can be done by detecting high fan-out and high ambiguity states in the ϵ-free IAM and by maintaining a mapping of the model states

Program	Workload
cat	Concatenate 38 files totaling 500 MB to a file
htzipd	Service 500 client requests simultaneously, transferring 152.2 MB of data in total
lhttpd	Service a single client request, transferring 151.7 MB of data
gnatsd	Service 2000 commands requested by a client

Table 3. Workloads.

Program	Unmonitored	Base	Monitored	%
cat	58.95	59.14	59.04	0
htzipd	13.22	13.35	14.46	8.3
lhttpd	29.17	28.82	29.08	0.9
gnatsd	35.61	36.29	34.36	0

Table 4. Program runtime in seconds.

Program	Code	Unoptimized Automaton	%	Compacted Automaton	%
cat	1,232	3.44	0.2	0.8	0.1
htzipd	1,840	132.3	7.1	77.1	4.1
lhttpd	1,888	5.9	0.3	4.3	0.2
gnatsd	1,992	32,243	1,618	8,966	450

Table 5. Memory usage in KB.

	% Runtime Overhead			% Memory Overhead		
Program	Dyck	VPStatic	IAM	Dyck	VPStatic	IAM
cat	56	32	0	49	194	0.2
htzipd	135	97	8.3	38	183	7.1

Table 6. Comparing models.

to program points to help identify instrumentation sites in program text. Automating the selection of instrumentation sites and evaluating its impact on model size is part of future work.

7. Evaluation

Program models for on-the-fly intrusion detection can be evaluated on two criteria: accuracy and efficiency. Greater accuracy makes these models useful by reducing false negatives and increased efficiency makes them usable by reducing time and space overheads. The IAM model has a runtime efficiency equal to that of a NFA model, which is the most efficient model possible.

We demonstrate the efficiency of our model by testing it with the four real-world programs shown in Table 1. For comparison purposes, we have chosen programs used in the literature. Tests were run on a Sun V100 550MHz UltraSPARC II with 256MB of RAM and running Solaris 9. Table 3 shows the workloads used in testing. Table 4 shows the runtime overhead for our model. Runtime is measured using the UNIX `time` utility. Time measurements are calculated over several runs. The base runtime represents the cost of library interposition and the monitored runtime includes the cost of operating the automaton. The monitored runtime does not include the setup time needed to load the program model from the disk (except in the case of `cat`). The difference between the base runtime and the monitored runtime represents the model operation overhead. The percentages compare this overhead against the base runtime. We attribute the slight variations between expected times (because of interposition and monitoring) and actual times to measurement noise. Except for `htzipd`, the runtime overhead for the programs is negligible. We are confident that the runtime overhead for `htzipd` can be reduced by using deterministic markers. Note that our solution scales efficiently to `gnatsd` which is a 32K program.

Table 5 shows the memory usage of the programs. The unmonitored memory usage of the code is obtained using the `pmap` command, which displays information about the address space of a process. The percentages compare the

automaton overhead against the unmonitored memory usage of the code. The automaton overhead is significant for `gnatsd` when compared to others. But this can be reduced by as much as 72.19% by including ϵ-transitions and delta successor states as described in Section 5. Also, note that the same automaton can be used if multiple instances of the program are running at the same time. Table 6 compares the runtime and memory overheads of the Dyck, VPStatic, and IAM models for common test programs. The IAM model is clearly more time and space efficient for these programs.

7.1. Discussion

In this paper, we demonstrated the efficiency of the IAM model by monitoring library calls instead of system calls as done in previous work. This choice is motivated by pragmatic considerations. Analyzing the source code of C libraries is a challenging task [17] (other approaches typically analyze statically-linked binaries [3, 4, 1]). The static analysis infrastructure used in our prototype was simply not able to handle these libraries; we plan to address this problem in future work. But, we also believe that switching to system calls will not affect our results. Library functions give a finer grained program model as they are usually more frequent. It is thus likely that overheads reported here are an upper bound on the costs of the approach. However, from an effectiveness perspective, monitoring the library interface alone is not sufficient for intrusion detection.

There are several approaches to handling recursion. The simplest solution is of course to allow imprecision at recursion points in the model based on the assumption that the actual loss of accuracy is small. The implication of this to mimicry attacks has to be considered. Furthermore, recursion is only a problem if there are library calls in the unwinding phase (i.e. if a library call is reachable in the control flow graph between the recursive call site and the function's exit), if not the attacker would gain absolutely nothing by following impossible paths. Thus, if a function `g()` is recursive and has library calls in an unwinding path, its calls can be transformed into guarded calls in the hybrid model. However, this can have an impact on the performance (which is why the Dyck model does not instrument recursive calls). None of the existing approaches, ours included, demonstrate an efficient way of handling recursion.

Our implementation has targeted C programs and extending it to object-oriented languages with dynamic binding raises concerns for accuracy and scalability. This is because, in C++ for example, virtual methods are invoked through function pointers and thus we would have to inline all possible implementations of the method at every call site. Static program analysis techniques can help. Experience with Java programs suggests that upwards from 90% of call sites can be devirtualized [15], *i.e.* it is possible to determine unambiguously which implementation will be invoked.

8. Limitations

The existing approaches to anomaly detection address only a part of the IDS problem: accurate and efficient monitoring of system call sequences. This is the simplest, yet important, concern for intrusion detection because an attacker has to use system calls to interact with the underlying operating system to cause harm (with the possible exception of DoS attacks). By accurately modeling the acceptable sequences of system calls, the models limit the attacker to only those expected system call sequences. However, there are several limitations that diminish the precision of these models.

Path Sensitivity. All the proposed models, ours included, treat branches in a conservative manner without evaluating the branch predicates because it needs more sophisticated static and dynamic program analysis. Such a path-insensitive modeling can be exploited by an attacker as illustrated in [1].

Granularity of Events. Granularity of events is a troublesome issue. Ideally, an IDS would monitor every single statement executed by the target program and validate each machine instruction. Clearly this is not possible and existing models are approximations at different levels of granularity. The coarser the approximation, the easier it is to mount a mimicry attack. So for example, restricting the observable events to system calls, means that library function calls are not captured in the model. Yet, library functions are common entry points for attackers because of their susceptibility to buffer overflow and format string vulnerabilities. Some vulnerable library functions such as the `string` family of functions do not make any system calls. Thus, a coarse model may not be able to observe the deviant behavior at the library interface and would have to rely on an out-of-sequence system call to detect an intrusion.

Library interposition techniques, such as the one used in this paper, allow to monitor the library functions called by a target program. However, an IDS based solely on library interposition will not be effective because if an attacker manages to exploit a vulnerability without setting off the IDS then he can further evade the IDS by directly issuing system calls in the "attack code." Therefore, coupling library interposition with kernel-level system call interposition is necessary. Such a combined IDS appears feasible, it requires two supervisors that must match up; one supervisor will monitor

library calls and the other system calls. A single program model can be used for both. A detailed discussion of this issue is beyond the scope of this paper.

Data flow Analysis. Data flow support is another requirement for more robust IDSs. It is well documented [17, 4] that even a naive approach that incorporates data flow by looking at arguments with constant values can dramatically improve the accuracy of models. To protect against mimicry attacks, it may be necessary to have more powerful predicates about the values of arguments. As an example, consider the case of a call that opens a file; if the leading part of the file name can be determined statically (even though the full name is constructed dynamically) then an IDS could prevent attempts to open files outside of the intended directory. Such predicates can be obtained by program analysis, but are likely to increase the runtime costs of monitoring, which is further reason to keep the costs of the basic program model low.

9. Conclusions

We have proposed an efficient and scalable solution to the problem of constructing conservative approximations of legal program behaviors for the purpose of host-based intrusion detection. Our approach based on an inlined automaton model (IAM) is context-sensitive and does not suffer from false positives. Constructing a basic IAM is simple and the resulting model is easy to understand. The overhead of monitoring programs based on an IAM is low and thus suggests that this technique could be deployed in production environments. The IAM construction algorithm has been shown to scale to a 32K line program with a substantial space overhead. We then show how to reduce this overhead with automaton compaction techniques.

Acknowledgments

We thank Barbara Ryder and Atanas Rountev for providing the PROLANGS Analysis Framework (PAF) used to implement our prototype and answering our questions. We also thank the authors of [1] for providing us with the `htzipd` source code and answering questions about their work. Finally, we thank the anonymous referees for their suggestions. This work was supported by sponsors of CERIAS, which is gratefully acknowledged, and in part by grant NSF TC #0209083.

References

[1] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, May 2004.

[2] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, May 2003.

[3] J. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, August 2002.

[4] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *11th Annual Network and Distributed Systems Security Symposium*, February 2004.

[5] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[6] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 2nd edition*. ACM Press, 2001.

[7] A. Jones and Y. Lin. Application intrusion detection using language library calls. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)*, 2001.

[8] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference (ACSAC)*, 1994.

[9] B. Kuperman and E. H. Spafford. Generation of application level audit data via library interposition. CERIAS TR 99-11, COAST Laboratory, Purdue University, Oct. 1998.

[10] T. Lane and C. E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security*, 2(3):295–331, 1999.

[11] W. Lee, S. J. Stolfo, and K. W. Mok. A data mining framework for building intrusion detection models. In *IEEE Symposium on Security and Privacy*, 1999.

[12] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, 2001.

[13] K. Tan, K. Killourhy, and R. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Recent Advances in Intrusion Detection (RAID)*, 2002.

[14] K. Tan, J. McHugh, and K. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Fifth International Workshop on Information Hiding*, 2002.

[15] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA)*, 2000.

[16] D. Wagner. *Static Analysis and Computer Security: New Techniques for Software Assurance*. PhD thesis, University of California, Berkeley, 2000.

[17] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[18] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *Ninth ACM Conference on Computer and Communications Security*, 2002.

[19] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Third International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2000.