CERIAS Tech Report 2006-12

IMPROVING SOFTWARE ASSURANCE USING LIGHTWEIGHT STATIC ANALYSIS

by Rajeev Gopalakrishna

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

IMPROVING SOFTWARE ASSURANCE USING LIGHTWEIGHT STATIC ANALYSIS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Rajeev Gopalakrishna

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2006

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

To one and all

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABBREVIATIONS

CFG	Control Flow Graph
CLI	Complete Likely Invariant
DBF	Dynamic Branching Factor
DFA	Deterministic Finite Automaton
EAM	Event Automaton Model
IAM	Inlined Automaton Model
LOC	Lines of Code
MDS	Misuse Detection System
NFA	Nondeterministic Finite Automaton
NLE	Non-Local Evidence
PDA	Push Down Automaton
PLI	Partial Likely Invariant
SBF	Static Branching Factor

ABSTRACT

Gopalakrishna Rajeev. Ph.D., Purdue University, May, 2006. Improving Software Assurance Using Lightweight Static Analysis. Major Professors: Prof. Eugene H. Spafford and Prof. Jan Vitek.

Software assurance is of paramount importance given the increasing impact of software on our lives. This dissertation describes research that explores two techniques to improve software assurance: a runtime approach in the context of host-based misuse detection systems (MDSs) and a compile-time approach to detect unknown software defects.

Host-based MDSs attempt to identify attacks by discovering program behaviors that deviate from expected patterns. We focus on automated and conservative misuse detection techniques. We present a static analysis algorithm for constructing a flow- and context-sensitive model of a program that allows for efficient real-time detection. Context-sensitivity is essential to reduce the number of impossible control-flow paths accepted by a MDS because such paths provide opportunities for attackers to evade detection. Our inlined automaton model presents an acceptable tradeoff between accuracy and performance in our experiments.

Static and dynamic approaches have been proposed over the years to detect security vulnerabilities. These approaches assume that the signature of a defect is known a priori. A greater challenge is detecting defects whose signatures are not known a priori—unknown software defects. We propose a general approach for detection of unknown defects. Software defects are discovered by applying data-mining techniques to pinpoint deviations from common program behavior in the source code and using statistical techniques to assign significance to each such deviation. We discuss the implementation of our tool, FaultMiner, and illustrate the power of the approach by inferring two types of security properties on four widely-used programs.

1 INTRODUCTION

A study released by the U.S Department of Commerce's National Institute of Standards and Technology (NIST) in 2002 estimated that software defects cost U.S economy $59.5 billion annually [1]. The study also found that over half of those defects are not found until late in the development cycle and that more than a third of the costs could have been eliminated if the defects had been identified and removed earlier in the cycle.

An error is a mistake made by a developer. It might be a typographical error, a misreading of a specification, or a misunderstanding of what a subroutine does [2]. An error might lead to one or more faults. Faults (also known as defects) are located in the text of the program. More precisely, a fault is the difference between the incorrect program and the correct version [2]. The execution of faulty code may lead to zero or more failures, where a failure is the (non-empty) difference between the results of the incorrect and correct program [2].

While the areas of software engineering, software quality, and software reliability have been studied extensively over the last three decades, they are mostly concerned with assuring the usability of software under normal conditions. Accidental failures resulting from accidental faults are their subject of concern and not malicious attacks resulting from vulnerability exploits. Software assurance has to deal with both accidental and malicious failures resulting from faults introduced either accidentally or deliberately [3]. It is this differentiating factor of intention that makes software assurance a challenging task.

A software vulnerability is an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy [4]. In other words, a defect whose execution can violate the security policy is a vulnerability. So all vulnerabilties are defects but not all defects are vulnerabilties. Triggering of defects leading to failures is accidental in nature but triggering of vulnerabilities leading to security violations or misuse may be deliberate and malicious. A malicious misuse

can, in general, do more harm than an accidental failure. Some of the losses may also be intangible such as the reputation of both the software developer whose software had the vulnerability and the software user who experienced the misuse.

Static analysis is the process of extracting semantic information about a program at compile time. This research focuses on improving software assurance using lightweight static analysis. It focusses on using static analysis to answer the following two questions: *is the software artifact exhibiting correct behavior at runtime?* and *will the software artifact exhibit correct behavior at runtime?* Answering the first question provides assurance on a continuous basis at runtime and answering the second question gives us assurance about the software artifact at compile time.

1.1 Misuse Detection

To Err is Human: as long as programming involves human activity, software vulnerabilities will exist. An attacker can subvert an executing instance of a software artifact by exploiting vulnerabilities present in it. This necessitates runtime monitoring of a software artifact to determine if it is exhibiting correct behavior at all times—an answer to the first question. A real-time host-based anomaly misuse detection system (MDS) attempts to identify in real-time if program behavior deviates from the known normal behavior. Static analysis techniques can be used to construct conservative models of program behavior that are guaranteed not to exhibit false positives. The challenge however is in balancing the important concerns of *accuracy*, as measured by the number of false negatives; *scalability*, the size of programs that can be handled by the monitoring algorithm; and *efficiency*, the runtime overhead of monitoring.

Current static analysis based approaches use sequences of system calls as characterization of program behavior [5–9]. Models are constructed in a flow- and context-sensitive manner to improve accuracy. However, this imposes an additional runtime overhead which in some cases is more than 100% [5, 7]. This is unacceptable for real-time misuse detection. We propose a new abstraction of program behavior known as the *Inlined Automaton*

Model (IAM) that not only retains the accuracy of the other models but also imposes negligible overhead [10]. We have implemented a prototype of an IDS based on this model and empirically evaluated its efficiency and scalability. The key idea behind this model is that space is traded-off for time. This is usually acceptable because time is the most important concern in real-time misuse detection systems. However, we have also developed automata compaction techniques to reduce the space-overhead of IAMs for operating in memory-constrained contexts.

1.2 Software Vulnerability Detection

A program's compliance with a property/policy can be checked either at run-time (dynamic checking and conventional testing) or at compile-time. The ability to check program properties without having to execute the program is especially appealing for security properties—properties that affect the security policy. A variety of static analysis checkers for detecting software vulnerabilities have been proposed [11–14].

Considerable research has looked at using program verification techniques to improve software assurance [11, 12, 14–21]. In this approach, given a specification of correct behavior in terms of program invariants, we can verify statically if there is any path in the program where the invariants do not hold. If so, it is indicative of an error. Manually specifying invariants is possible only when we know what the invariants are. Certain language-specific invariants such as avoiding the dereferencing of null-pointers and avoiding memory leaks, and operating-system related invariants such as sanity-checking user-pointers before dereferencing them in the kernel are well known. There is considerable prior research in this area of finding *known defects*—software defects or vulnerabilities that result from a violation of known invariants. Examples include techniques to detect buffer-overflow bugs, format-string bugs, and file-system race conditions.

However, a program typically has several other invariants that are not language or operating-system specific but that are program-specific or particular to the program logic or semantics. Such invariants are rarely documented but are nonetheless critical for the

program's correct behavior. The challenge is in finding such invariants because what may be determined as invariants might simply be coincidences. For this reason, they are referred to as *likely program invariants*. Once we know the likely invariants, the problem reduces to that of program verification. Defects that result from a violation of such likely invariants can be thought of as *unknown defects* because the invariants are not known a priori. Recently, several approaches have been proposed to extract likely invariants from a program. Broadly, they fall into two categories: dynamic approaches [22–26], which observe a program's runtime behavior, and static approaches [27–29], which analyze program text to detect likely invariants. Static approaches are appealing because they have the advantage of observing all the program paths. The current static approaches to finding unknown defects consider simple temporal invariants, in specific contexts, and use ad-hoc techniques. We propose a general approach to finding unknown defects by considering temporal invariants on general *events* and by using techniques derived from well-known algorithms from the data-mining literature to determine likely invariants. This research compliments the work on finding known defects in answering the second question of whether the analyzed software artifact will exhibit correct behavior at runtime.

1.3 Thesis Statement

This dissertation describes the work done to validate the following hypothesis:

It is possible to build models of software artifacts using lightweight static analysis that can be used to efficiently detect and avoid misuse at runtime and detect unknown defects at compile-time.

For the purposes of this dissertation, *lightweight static analysis* refers to control-flow analysis, which is typically easier than data-flow analysis. The remaining terminology is introduced later in appropriate sections.

1.4 Organization

The dissertation is organized as follows: Chapter 1 has described the broad motivation for this research in the context of related work and presented the thesis statement. Chapter 2 describes the research performed to validate the first part of the hypothesis by demonstrating that we can build models of software artifacts that can be used in runtime monitoring to efficiently detect and avoid misuse. Chapter 3 describes the research done to validate the second part of the hypothesis by building models of software artifacts that can be used to detect unknown defects at compile-time. A detailed discussion of related work is presented separately for the two parts in the above chapters. Finally, Chapter 4 presents the conclusion, summarizes the major contributions, and outlines directions for future work.

2 EFFICIENT INTRUSION DETECTION USING AUTOMATON INLINING

The goal of a host-based *misuse detection system* (MDS) is to identify an attacker's attempts to subvert processes running on the system. An anomaly-based MDS achieves this by identifying program behaviors that deviate from the known normal behavior. Intuitively, MDS 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*) [30–36], by a domain expert who creates a specification of the program (*manual analysis*) [37], or by automatically creating a specification of the program using static program analysis [5–9]. All approaches must deal with false positives, when the MDS 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 MDS, 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. However, this conservativeness introduces inaccuracies in the models that can potentially lead to false negatives. The accuracy of these approaches is illustrated in Figure 2.1.

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 con-

siders the order of execution of statements in the program. The basic model described by Wagner and Dean [9] 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 using 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, we 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 [9], 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 [9], 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 [5, 7] 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 misuse

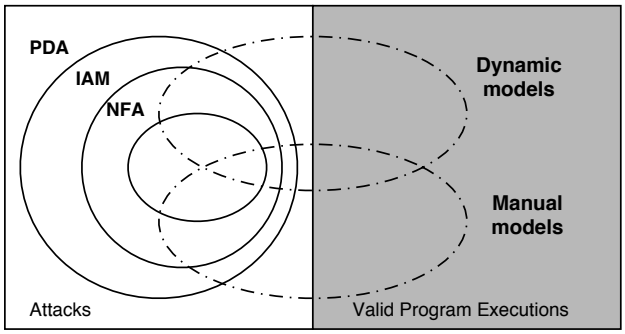

Figure 2.1. Accuracy of host-based misuse detection models. The figure shows program traces indicating attack and valid executions. Both dynamic and manual models flag some valid traces as erroneous (false positives) and miss some invalid traces (false negatives). Automatically constructed models based 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.

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 [38–40]). To make such attacks more difficult to carry out, misuse 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 MDSs but also increase the amount of state needed for verification and thus further increase runtime costs.

We present a new abstraction of program behavior referred to as an *Inlined Automaton Model* (IAM) that is as accurate, in the absence of recursion, as a PDA model and at least 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 misuse detectors.

The remainder of the chapter is organized as follows. Section 2.1 describes existing approaches to statically-constructed model-based anomaly detection. Sections 2.2 and 2.3 describe the construction of IAM and Sections 2.4 and 2.5 describe implementation issues and experimental results. Section 2.6 discusses the concept of deterministic markers. Automata compaction techniques are described in Section 2.7. The efficiency and effectiveness aspects of our MDS are discussed in Sections 2.8 and 2.9. Section 2.11 discusses the challenges faced by existing approaches and Section 2.12 describes other related work.

2.1 Statically-generated Model-based Automated Misuse 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 MDS is by Wagner and Dean [9]. 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 2.3). Finally, the abstract stack model eliminates such impossible paths by modeling the call stack of a program using a PDA. However, [9] demonstrates 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.* [6] 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 invo-

cations 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 for 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 [5, 7] 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 [5] is a statically-constructed variant of the dynamic context-sensitive VtPath model [30]. 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.

Lam and Chiueh [41] propose a context-sensitive system-call level MDS based on graph inlining called *Paid*. They insert *notify* calls to convert the inlined model into a deterministic finite state automaton (DFA) and also to handle function pointers and non-standard control flow caused by `setjmp/longjmp`. They report performance overheads between 2%-7% for network daemons such as `wu-ftpd` and `sendmail`. We came up with our approach independently in [10] without any knowledge of this work.

2.2 The Inlined Automaton Model

```
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 2.2. An example program.

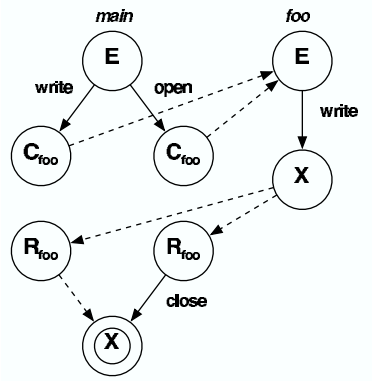

Figure 2.3. NFA representation of the program in Figure 2.2. E, X, C, and R represent entry, exit, call, and return nodes respectively. The dotted lines represent ϵ-transitions in the NFA.

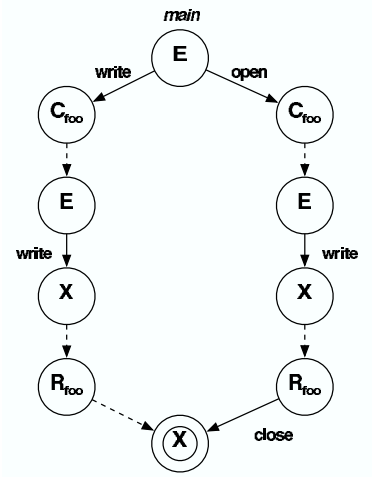

Figure 2.4. The ϵ-IAM representation of the program in Figure 2.3. E, X, C, and R represent entry, exit, call, and return nodes respectively. The dotted lines represent ϵ-transitions.

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 correspond-

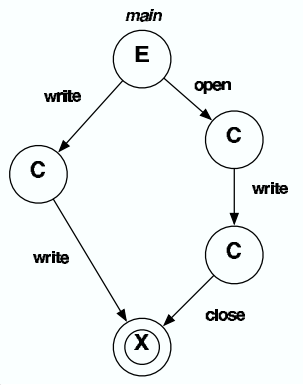

Figure 2.5. An ϵ-free IAM representation of the program in Figure 2.3. E, X, and C represent entry, exit, and call nodes respectively.

ing 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 2.3 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 2.3, 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 2.2.1 *Formally, an ϵ-NFA N for a program P is represented as $N = (Q, \Sigma, \delta, q_0, F)$ [42] 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 $q \in Q, \exists\, T$ such that $T(q) \in \{E, X, C, R\}$, 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 $\exists\, l \in \Sigma \cup \{\epsilon\}$ such that $\delta(q, l) = s$. *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 $\exists\, l \in \Sigma \cup \{\epsilon\}$ such that $\delta(s, l) = q$. *Fan-in* of state q is defined as the cardinality of the set *predecessor*(q).

The IAM representation of the program in Figure 2.4 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 2.2.2 *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 2.5 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 [6] decided not to pursue this approach. The state space can be somewhat limited by

```
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 2.6. A recursive program.

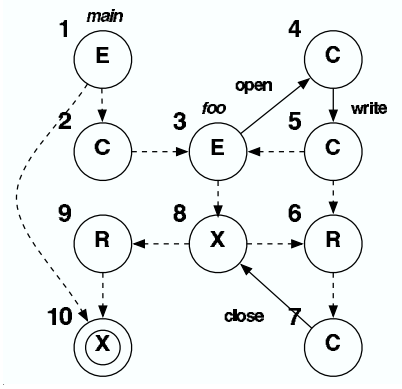

Figure 2.7. IAM representation of the recursive program in Figure 2.6. Dotted lines representing ϵ-transitions have been retained for clarity. The node sequence 1-2-3-4-5-3-8-9-10 which translates to the system call sequence `(start, open, write, end)` is an impossible path.

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 2.7 discusses space compaction techniques.

```
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 2.8. An indirect-recursive program. The program does exactly the same thing as the program in Figure 2.6 but using mutually recursive functions foo1() and foo2().

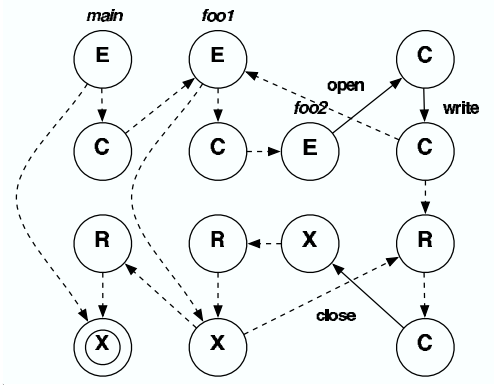

Figure 2.9. IAM representation of the program in Figure 2.8. The dotted lines representing ϵ-transitions have been retained for clarity.

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 2.7 and 2.9. Recursion introduces impossible paths, for example in Figure 2.7, 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 a system-call level IAM to the formalization of [5].

Theorem 2.2.1 *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 [5], then we have $L(PDA(P)) \subseteq L(IAM(P))$.*

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

2.3 Monitoring Programs with IAM

Our current implementation of IAM monitors library function calls. The runtime monitor is implemented as a library interposition mechanism [43]. It intercepts calls to library functions and checks them against the model. Figure 2.11 gives pseudo-code for the monitoring algorithm. 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.

2.3.1 Monitored Events

It should be noted that the algorithm monitors possibly more events than other approaches because we track library functions irrespective of whether they make system calls or not. In [10], we hypothesized that this would generally result in more states and more transitions in our automaton and that this bigger size increases the runtime overhead because of the greater search space. Therefore, modeling libraries instead of system calls would be a

```
node {
  unsigned int nodeid : 20
  unsigned int funid  : 11
  unsigned int succ   :  1
}

int funid
node[] curr
node[][] model
```

Figure 2.10. Data structures. `nodeid` is the node identifier. `funid` is the library function identifier. The `succ` bit indicates if a node is the last node or not in the list of successor nodes. `curr` is a vector of current nodes.

```
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 2.11. IAM monitoring algorithm.

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 Solaris). 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 measure-

ments presented in Section 2.5 can be considered an upper bound for an implementation of a similar approach based on system calls. Section 2.8 has more discussion on this issue.

2.3.2 Handling Non-standard Control Flow

Function pointers, `setjmp/longjmp` primitives, and signals have to be handled to obtain a sound model. Failure to do so will result in false positives.

1. Function Pointers

   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 problem can be solved by either ignoring function pointers completely or by combining the construction of control-flow graph with pointer analysis. Ignoring function pointers is unsound. In [10], we resolved function pointers to all defined functions with the same number and type of arguments as the function pointer invocations. Although this was 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.

   Wagner and Dean resolve all function pointers to every function that has its address taken. They used manual annotation in some cases to refine the targets. Giffin et al. generate models from weakly typed binary code. Thus they use a combination of data flow analysis, manual annotation, and resolving to functions whose address is taken. Lam and Chiueh use a runtime technique to resolve a function pointer call to all the functions whose address is taken using their *notify* calls.

   Our current implementation resolves function pointers to functions whose address is taken and that have the same number and type of arguments as the function pointer invocation. Future work should incorporate precise pointer analysis to accurately

resolve function pointers. This would significantly decrease the model size for programs with increased function pointer usage especially in the case of IAM which uses inlining. This would also improve the precision of MDS by reducing non-determinism at function pointer invocations and therefore reduce the likelihood of mimicry attacks.

2. `setjmp` and `longjmp` Primitives

   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. They are used in error routines and interrupt handlers to go to a safe state.

   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 connected a longjmp call to every setjmp call in the control flow graph in [10].

3. Signals

   Signals are used extensively in privileged programs and network daemons. In [10] we identified the signal handlers in a program and constructed separate context-sensitive models for them. Signal handlers can be invoked asynchronously at any program point where signals have been enabled. Including models of signal handlers as part of the program model at every point where signals are enabled would be unreasonable. [8, 9, 41] describe runtime techniques that can be used to used to infer the start and end of a signal event that can then be used to modify the model at runtime.

2.3.3 Recursion

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. None of the existing approaches, ours included, demonstrate an efficient way of handling recursion.

2.4 Implementation Details

In [10], we implemented the model generation prototype using the PROLANGS Analysis Framework (PAF) from Rutgers University [44]. The current implementation uses CIL [45]. CIL (**C I**ntermediate **L**anguage) is a high-level representation along with a suite of tools that facilitates whole program analysis of C programs.

2.4.1 Model Generation

The `Makefile` of a software application is modified to invoke CIL instead of GCC. CIL processes the source files of the application based on the options specified and then invokes GCC ultimately. We use the `--merge` option of the CIL driver to collect all the source files that make up the application into one single source file. We also use the `--keepmerged` option to make CIL save the merged source file. The model generator that is implemented as a CIL module is invoked as a command line argument on this merged source file. This generates NFAs for every user-defined function in the application along with function pointer target information. This data is then used to construct an IAM.

2.4.2 Compiler Directives and Optimizations

GCC provides several built-in functions such as `__builtin_constant_p`, `__builtin_va_start`, and `__builtin_va_end` [46]. Built-in functions are compiler directives that can be used to perform a variety of tasks. `__builtin_constant_p`, for

example, can be used to determine if a value is a compile-time constant. CIL includes such directives in the merged file because GCC has not yet been invoked. Therefore we have to deal with such compiler directives during model generation. We exclude such built-in functions from the model because they are not interposable at the library interface. But GCC has also built-in versions of several library functions. These are intended for optimization and in cases where they are not or cannot be optimized, such built-in functions invoke the library functions. We include library functions called by such built-in functions in our models irrespective of whether they will be optimized or not. We have not seen a mismatch between the generated models and the binaries for our benchmarks and workloads because of this.

Model generation should either mimic the compiler optimizations or be invoked after optimizations have been performed so that the program models accurately represent runtime behavior. For example, we excluded from the models `strlen` functions whose arguments were string literals. Such calls are optimized (and hence not present in binaries) because they can be computed at compile-time.

2.4.3 Library Interposition

Library interposition is "the process of placing a new or different library function between the application and its reference to a library function" [47]. Most applications make calls to library functions that are part of shared libraries such as LIBC. Library interpositioning makes use of the dynamic linking feature to intercept calls to such functions. This can be achieved using the `LD_PRELOAD` environment variable. The intercepting function can perform a variety of tasks before calling the actual library function. It effectively behaves as a "wrapper" function. A detailed description of this technique can be found in [47, 48].

We use the interposed functions to check against the model if the issued function call is acceptable at the model's current state. If so, we let the interposed function invoke the actual library function. If not, we raise an alert and terminate the process that issued the function call. Figure 2.12 illustrates the working of our prototype.

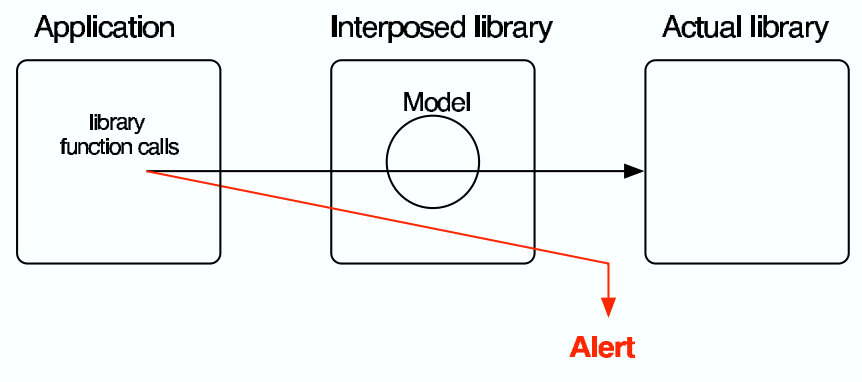

Figure 2.12. A model-based MDS based on library interposition. The library function calls issued by an application are intercepted by interposed libraries and checked against the application's model. If the issued calls are considered acceptable in the model, the interposed functions call the actual library functions. If not, an an alert is generated.

2.4.4 Interposing Variable-argument Functions

Variable-argument functions in C such as the `printf` and `execl` family of functions present an interesting challenge while interposing them. For such functions, the callee has to determine the number of arguments that should be read off the stack at runtime. Interposed versions of such functions have to reconstruct the argument list before calling the actual function.

We have achieved this in two ways. For the `printf` family functions, we process the variable-arguments using the `va_start` and `va_end` routines to construct a `va_list` variable and then call the corresponding `vprintf` versions that use `va_list`. For the `execl` family functions that do not have `va_list` versions, we traverse the stack, construct the array of arguments, and call the corresponding `execv` versions that use argument arrays. Information on these functions can be obtained from their manual pages using the `man` command.

Table 2.1
Test programs.

Program	Software Version	LOC	Description
htzipd	LiteZipper-0.1.6	6,839	A proprietary HTTP server implementation
lhttpd	lhttpd-0.1	819	A fast and efficient HTTP server capable of handling thousands of simultaneous connections
wu-ftpd	wu-ftpd-2.6.2	26,317	A widely-used ftp daemon
gnatsd	gnats-4.0	55,778	The server component of GNATS, which is a set of tools for tracking bugs reported by users

2.4.5 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 2.10). 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 32 bits to conserve space. The 20 bits and 11 bits bit-fields used for the nodes are sufficient to represent our test programs.

2.5 Evaluation

Program models for on-the-fly misuse 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. The addition of markers further improves the efficiency by reducing non-determinism and has the potential to ex-

Table 2.2
Characteristics of IAM models.

Program	ϵ-IAM		ϵ-free IAM	
	states	transitions	states	transitions
htzipd	59,454	91,384	24,273	137,159
lhttpd	715	990	401	1,008
wu-ftpd	315,950	1,210,826	169,763	3,042,261
gnatsd	1,572,319	2,577,256	221,481	19,080,312

hibit the runtime characteristics of a DFA (c.f. Section 2.6). It also improves the accuracy of the model (c.f. Section 2.6.3).

We demonstrate the efficiency of our model by testing it with the four real-world programs shown in Table 2.1. Table 2.2 gives the basic characteristics of the IAM models for these programs. Tests were conducted on an isolated network with the servers and MDS running on a 2.8 GHz Linux machine with 1 GB RAM and the clients running on a 597 MHz Linux machine with 128 MB RAM.

Table 2.3 shows the workloads used in testing. Table 2.4 shows the runtime overhead for our model. Runtime is measured using the UNIX `time` utility. Time measurements are calculated over ten runs and the best time is considered. The delays in the other runs are assumed to have been caused by interference from the underlying hardware and software components. The monitored runtime does not include the setup time needed to load the program model from the disk.

From Table 2.4, we see that monitoring using IAM does not add significant overhead (1.5% to 2.5%) except in the case of `gnatsd` for Workload-Two where the overhead is 325.27%. This is reduced to 31.12% on using deterministic markers in the model as shown in Section 2.6. Workload-Two replaced 300 `help` commands in Workload-One (used in [10]) with `subm` commands. The `subm` command adds a problem report to the `gnatsd`

Table 2.3
Programs and workloads.

Program	Workload
htzipd	Transfer 171.28 MB of data to a client
lhttpd	Transfer 171.28 MB of data to a client
wu-ftpd	Transfer 171.28 MB of data to a client
gnatsd-Workload-One	Service 2000 commands
gnatsd-Workload-Two	Service 2000 commands (includes 300 `subm` commands)

Table 2.4
Effect of monitoring on performance.

Program	Unmonitored Time	Monitored Time	Percentage Overhead
htzipd	26.56s	26.95s	1.47%
lhttpd	16.16s	16.56s	2.48%
wu-ftpd	18.44s	18.73s	1.57%
gnatsd-Workload-One	46.91s	47.59s	1.45%
gnatsd-Workload-Two	34.58s	147.06s	325.27%

database. The code for `subm` command is more complex than `help` and other commands used in Workload-One.

2.6 Deterministic Markers

Non-determinism can be interprocedural or intraprocedural. Interprocedural non-determinism arises in NFA-based models when legal transitions to multiple call sites are possible at a function's exit node. Context-sensitive models do not have this problem. Intraprocedural non-determinism is caused by conditionals and loop constructs in program text. This is a

result of path-insensitivity in the models. If at points of such non-determinism, the observable events are the same among the possible branches, it leads to ambiguity. The monitor then has a set of possible current states. This not only imposes additional performance overhead but also introduces imprecision in the model that can be exploited in a mimicry attack.

A high degree of intraprocedural 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 states as its current state and check successors for each of them at runtime.

We introduce the terms *static branching factor* (SBF) and *dynamic branching factor* (DBF) at this point. Static branching factor for a state is the number of successors of that state in the model. Dynamic branching factor at any instance during monitoring is the sum total of the number of successors of all the states in the model that are considered as current state by the MDS. DBF will be the same as SBF in the absence of ambiguity introduced by non-determinism. Note that DBF and SBF are different from the average branching factor metric that is used in [6, 7, 9] for measuring the precision of MDS models. We consider all the library functions while measuring SBF and DBF unlike the average branching factor metric, which considers only a subset of system calls deemed as dangerous.

We introduce the concept of *deterministic markers* as a solution to intraprocedural non-determinism. Deterministic markers are unique transition symbols introduced in the program text to reduce intraprocedural non-determinism and thereby decrease the DBF and the search space of the runtime monitoring algorithm. Conceptually, they are similar to the renaming and null call insertion techniques described in [5–7]. 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) in the case of intraprocedural non-determinism.

In [10], we used markers only for `gnatsd` along the paths exercised by Workload-One. Eleven sites were manually identified and null library calls were introduced. Below, we outline algorithms and rationale for automatically identifying locations for insertion of markers.

2.6.1 Types of Markers

We propose three types of markers based on program constructs that cause intraprocedural non-determinism:

1. Switch Markers

   Switch statements in C can be points of significant non-determinism. Programs such as network daemons that accept user commands have lexers and parsers that have to process several types of tokens and so use `switch` statements. Inserting a unique marker at the beginning of every `case` statement resolves the non-determinism that is present at such program points.

   Identifying `case` statements in program text is a simple task. While such markers can be conservatively inserted for all `case` statements, we can reduce the number of markers added by discarding `case` statements that do not make function calls in their scopes. Currently, we simply `grep` the program text for `case` statements and add a marker if it is not obvious that function calls are not made in their scopes. A `sed` script [49] can further reduce the effort in adding such markers.

2. Function Pointer Markers

   Function pointers that are identified as having multiple targets also can result in significant non-determinism depending on the number of targets. For example, in

`gnatsd`, at the point where the server determines the command it has to process, it invokes the appropriate command handler using a function pointer. That function pointer is determined as having 31 targets based on the number and type of arguments of functions whose address is taken. We introduce a unique marker at the beginning of every target function's body to resolve any non-determinism that is present at function pointer invocations.

The above two types of markers are ad-hoc in nature but work in practice. Table 2.5 shows the reduction in average DBF for `wu-ftpd` because of the addition of switch markers. These markers also had an unexpected effect of decreasing the model size. The model size for `wu-ftpd` decreases by 62.28% on inserting the switch markers. The reason for this is that although adding markers increases the number of states in the model, the number of transitions typically decreases. This is because multiple states that had a high fan-out now have only the marker as their single successor state and the marker is the only state that has the high fan-out. This is illustrated in Figure 2.13.

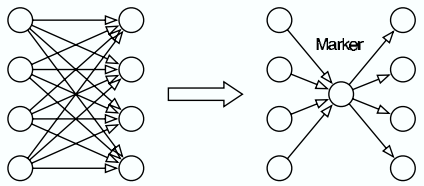

Figure 2.13. The effect of switch and function pointer markers on fan-out. Adding these markers increases the number of states in the model but the number of transitions typically decreases. Multiple states that had a high fan-out now have only the marker as their single successor state and the marker is the only state that has the high fan-out.

3. Library Markers

Ambiguity in the ϵ-free IAM model arises in states that have multiple transitions (to successors) for the same symbol. Library markers can be used to reduce or eliminate non-determinism. The algorithm for determining the locations of such

library markers runs in *O(n)*, where *n* is the number of states in the ϵ-free IAM. For each state, we identify common transition symbols whose successors have not yet been assigned markers. For successor states of all such transitions, except one, we assign a unique marker. Assigning unique markers to all but one successor with common transition symbols is sufficient to disambiguate the successors. We keep track by maintaining a list of locations in program text corresponding to the states that have been assigned markers. Based on the output of this algorithm, we insert unique library markers at the determined program locations.

In the current implementation, we can only reduce the non-determinism in the IAM but cannot convert it into a DFA using library markers. This is because we insert the markers in the program text and not in an inlined intermediate representation as is done in [41]. Consider the program text shown in Figure 2.14. The non-determinism is caused by calls to `printf` within the function `foo()` along the two branches of the conditional. This cannot be eliminated even if we insert a library marker before the call to `printf` in `foo()`. However, if we insert markers on an inlined intermediate representation then we can insert two unique markers at the two inlined locations of `foo` to resolve the non-determinism.

```
....
if (...) {
  foo("Hello World");
}
else {
  foo("Goodbye World");
}
....
foo(char *str) {
  printf("%s", str);
}
```

Figure 2.14. Example program where non-determinism cannot be resolved using library markers.

Table 2.5
Number of library calls intercepted and average dynamic branching factor (DBF).

Program	Library Calls Intercepted	Avg. DBF	Library Calls Intercepted with Markers	Avg. DBF with Markers
htzipd	184,828	29.32	–	–
lhttpd	526,520	1.67	–	–
wu-ftpd	88,863	27.09	88,882	2.99
gnatsd-Workload-One	73,975	6550.24	82,150	15.15
gnatsd-Workload-Two	196,144	11,589.96	243,531	3,077.67

We added 143 switch markers to `wu-ftpd` and 396, 31, and 44 of switch, function pointer, and library markers respectively to `gnatsd`. We added only those library markers that were suggested for `memcpy`, `realloc`, and `signal` functions because only these seemed to affect the DBF for our workloads. The process of adding markers can be iterative at model construction time. Instead of adding all the suggested markers, one can add only those markers that will be along frequently executed paths.

Table 2.5 shows the number of library calls intercepted and the average DBF for the different programs and workloads. DBF is summed up for the all states the MDS transitions to during the execution of the workload and the sum is averaged over the number of library calls intercepted to obtain the average DBF. We observe that adding markers reduces the average DBF for `wu-ftpd` and `gnatsd`. This helps in reducing the performance overhead because the number of successors to be checked is fewer. Table 2.6 shows the improvement in performance because of the addition of markers.

Another observation is that an average DBF of about 3077 for `gnatsd` Workload-Two results in 31.12% overhead but an average DBF of about 6550 does not introduce much overhead for `gnatsd` Workload-One. We reason that this is because of the different commands used in the two workloads. Workload-Two replaces 300 `help` commands

Table 2.6
Effect of markers on performance overhead.

Program	Unmonitored Time	Monitored Time	Monitored Time with Markers	Percentage Overhead
htzipd	26.56s	26.95s	–	–
lhttpd	16.16s	16.56s	–	–
wu-ftpd	18.44s	18.73s	18.28s	0%
gnatsd-Workload-One	46.91s	47.59s	47.17s	0.55%
gnatsd-Workload-Two	34.58s	147.06s	45.34s	31.12%

from the Workload-One with `subm` commands. The `subm` commands perform more work at the server and only inform the client that the report has been added whereas the `help` command prints out the help information to standard output at the client. So the bottleneck in the case of the `help` command is caused by I/O at the client where the timing is performed and this reduces the impact of the overhead caused by the high branching factor. This illustrates that the impact of the overhead from an MDS might differ based on the nature of application.

Figures 2.16-2.19 show the distribution of DBF across library functions intercepted for a single workload of the benchmarks. Because the number of intercepted library functions was significantly high to be plotted individually, we plot the average DBF over several functions. The plots for `wu-ftpd` and `gnatsd` are for versions with markers.

2.6.2 Space Complexity of DFA

We prove that we can convert the IAM into a DFA by adding n unique markers in the worst case, where n is the number of states in the IAM. The proof for this is given below. In practice, we add markers only when there is non-determinism at a state in the form of multiple transitions for the same symbol. So the size of the DFA constructed using

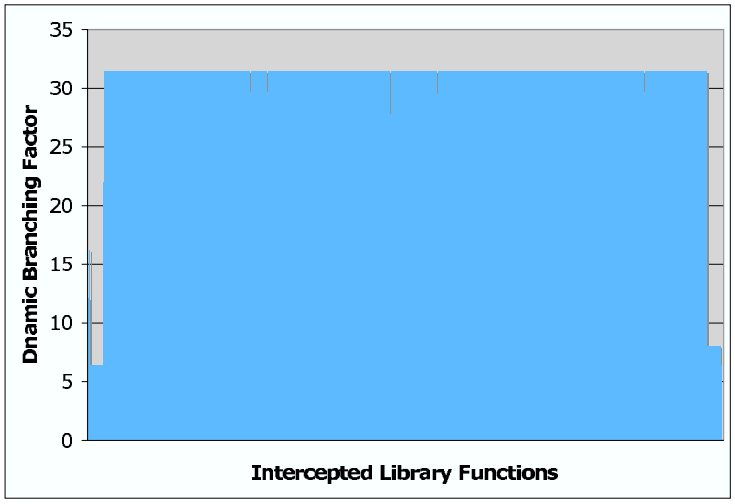

Figure 2.15. Distribution of DBF across library functions intercepted for a single workload of `htzipd`. At every library function intercepted by the MDS, DBF indicates the number of states in the model that are checked for possible transition.

markers is less than double the size of the IAM in terms of the number of states. Also, the number of transitions increases by at most *n*.

Corollary 2.6.1 *If N is an ϵ-free IAM for a program P given by $IAM = (Q, \Sigma, \delta, q_0, F)$ and $s_i \in Q$ then $\forall\ s_j \in$ predecessor(s_i), $\exists$ a single symbol l_i such that $\delta(s_j, l_i) = s_i$.*

The above corollary states that for each state in the ϵ-free IAM, all the incoming transitions have the same symbol. This is an artifact of the construction of the IAM from a program.

Theorem 2.6.2 *An ϵ-free IAM with n states can be converted to a DFA by adding n more states.*

Proof Let IAM be an ϵ-free IAM for a program P given by $IAM = (Q, \Sigma, \delta, q_0, F)$. Then $n = |Q|$. For every state $s_i \in Q$, we add a unique marker state m_i such that $\delta(m_i, l_i) =$

s_i, where l_i is the incoming transition symbol for s_i. Then $\forall\ s_j \in$ predecessor(s_i) except m_i, we replace $\delta(s_j, l_i) = s_i$ by $\delta(s_j, l_{m_i}) = m_i$, where l_{m_i} is the incoming transition symbol for m_i.

The resulting automaton has $2n$ states because we have added one unique marker for each of the n states. Every state s_i has outgoing transitions only to unique marker states along their respective unique marker symbols. And every marker state m_i has a single transition to its corresponding s_i along l_i. So no state in the resulting automaton has two outgoing transitions for the same symbol. Therefore it is a DFA with $2n$ states. Also, note that the resulting DFA has n more transitions than the IAM. These are the transitions connecting the markers to their respective states. ■

An example illustrating this proof is shown in Figures 2.20-2.22. Figure 2.20 shows an IAM with non-determinism in states s_0 and s_1 for the transition symbol c. Figure 2.21 shows how the addition of four markers, one for each of the states, can eliminate the non-

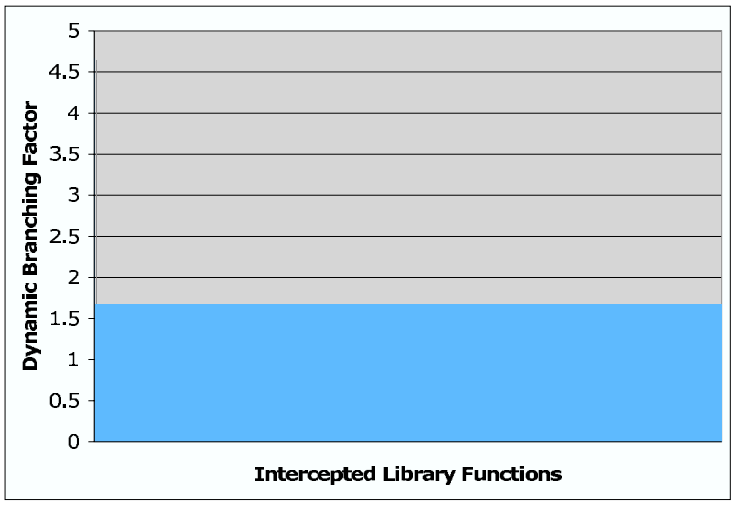

Figure 2.16. Distribution of DBF across library functions intercepted for a single workload of `lhttpd`. At every library function intercepted by the MDS, DBF indicates the number of states in the model that are checked for possible transition.

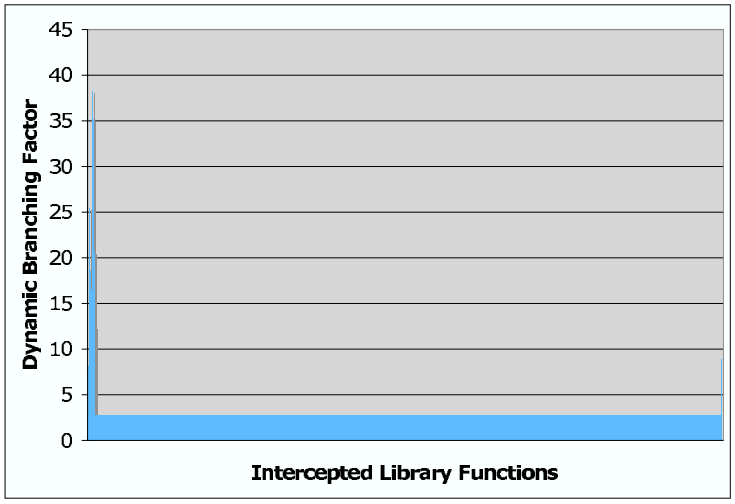

Figure 2.17. Distribution of DBF across library functions intercepted for a single workload of `wu-ftpd`. At every library function intercepted by the MDS, DBF indicates the number of states in the model that are checked for possible transition.

determinism. Figure 2.22 shows that in practise, the number of markers required is less than the number of states. In this case, we need only two markers and not four.

2.6.3 Markers and Mimicry Attacks

DFA construction using library markers has another benefit in that it can help prevent certain mimicry attacks that neither the IAM nor a DFA generated using *subset construction* [50] can. Consider the program text shown Figure 2.23. It performs access control on some user input and based on the result, it determines if a shell should be spawned or not. If access is denied, it performs some book keeping operations, such as logging, and exits. The `bookKeeping()` function does not call any library functions and has a buffer overflow vulnerability (caused by direct pointer manipulation for example) that can be triggered by some user input. So the user can execute the buffer over-

flow (on being denied access), cause `bookKeeping()` to return to the location where `execv(``/bin/sh'',``sh'')` is called, and thus subvert the access control mechanism.

Figure 2.24 shows a part of the IAM model for the program in Figure 2.23. An MDS that is state s_0 will transition to both state s_1 and s_2 on seeing `printf`. If the `printf` executed was the one where access was denied, the attacker can still overflow the `bookKeeping()` function and transfer control to `execv(``/bin/sh'',``sh'')`. `execv` is a legal transition from state s_2, which is one of the current states of the MDS. The MDS will therefore allow the subversion of access control without generating an alert. This is because of the imprecision in the model caused by non-determinism.

The DFA shown in Figure 2.25 is constructed using the subset construction algorithm described in [50]. This removes non-determinism in the model but it does not prevent the

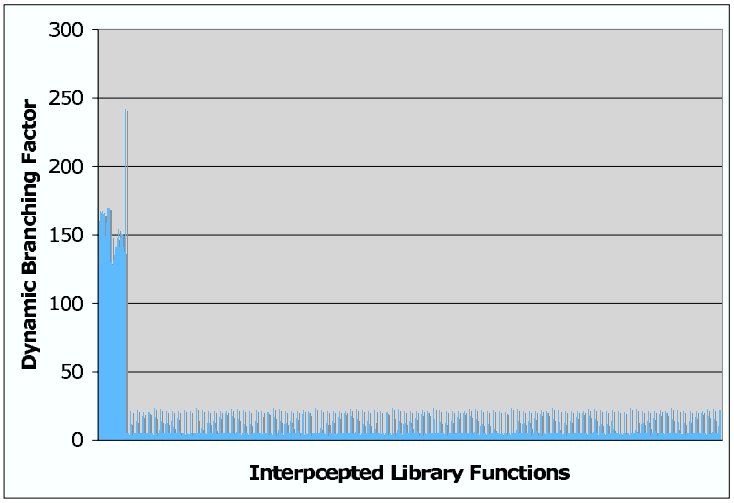

Figure 2.18. Distribution of DBF across library functions intercepted for `gnatsd`-Workload-One. At every library function intercepted by the MDS, DBF indicates the number of states in the model that are checked for possible transition.

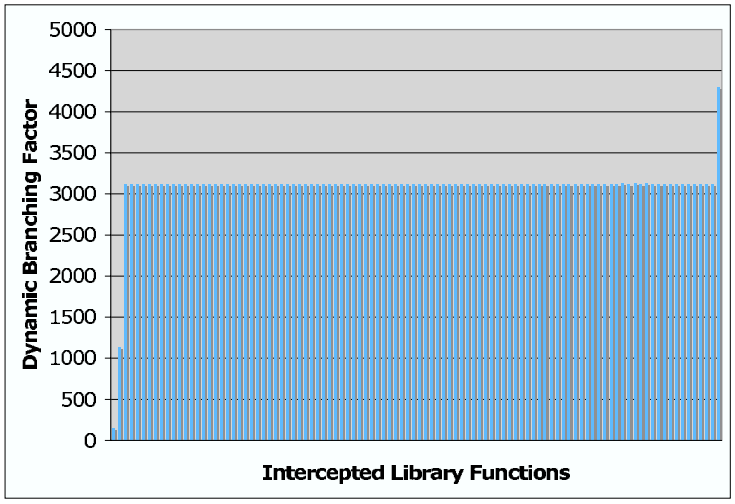

Figure 2.19. Distribution of DBF across library functions intercepted for `gnatsd`-Workload-Two. At every library function intercepted by the MDS, DBF indicates the number of states in the model that are checked for possible transition.

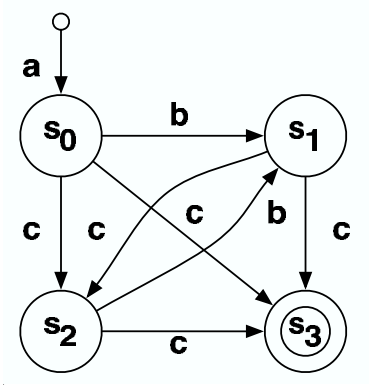

Figure 2.20. IAM model with non-determinism.

above mimicry attack. This is because the DFA removes non-determinism by merging states s_1 and s_2. The attacker can still call `execv` legally from this merged state.

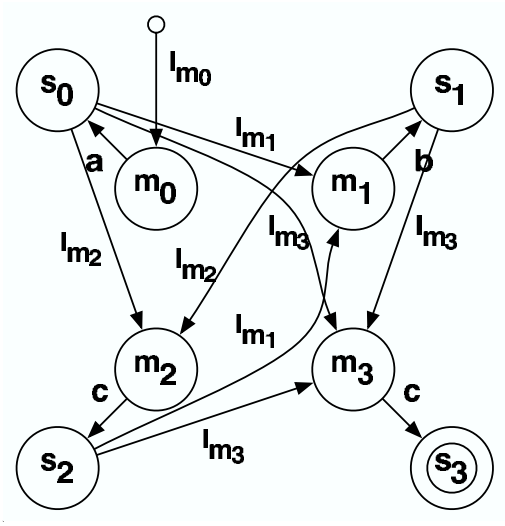

Figure 2.21. IAM model with four markers to eliminate non-determinism.

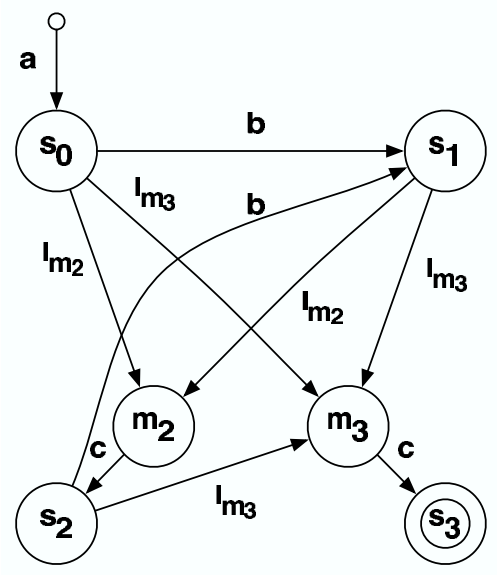

Figure 2.22. IAM model with two markers to eliminate non-determinism.

If we insert a unique marker in the program text before the `printf` call that precedes the `execv` call and then construct an IAM, we will obtain a DFA as shown in Figure 2.26. Notice that in this model, the marker disambiguates the two printfs and `execv` is allowed

```
....
allow = access_control_routine(user input);
printf("Access control decision made");
if (allow) {
        printf("Allow access");
        execv("/bin/sh","sh");
}
else {
        printf("Deny access");
       // Below function has no calls to library
       // functions but has a buffer overflow
        bookKeeping();
}
exit();
....
```

Figure 2.23. Example program that illustrates the mimicry attack possible when a model based on IAM or a DFA derived using subset construction is used. A model based on a DFA derived using library markers is not vulnerable to this mimicry attack.

only if it was preceded by a marker and a `printf`. If access was denied, the MDS would not see the marker and thus would allow only `exit` to follow the `printf`. So by constructing a DFA using markers, we can not only remove non-determinism but also prevent mimicry attacks such as the one described in the example, thus improving the precision of the MDS.

Wagner and Dean [9] suggest the use of a DFA to improve the runtime performance of MDSs. [41] constructs a DFA using `notify` calls which are similar to markers. But they do not describe the rationale behind using this approach instead of the well-known subset construction approach. The combination of superior time complexity and resistance to the above described mimicry attacks is indeed the reason why a DFA constructed using markers is better than a DFA constructed using the subset algorithm.

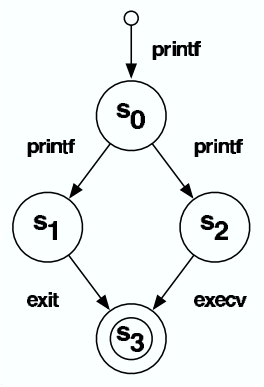

Figure 2.24. IAM model which is susceptible to a mimicry attack.

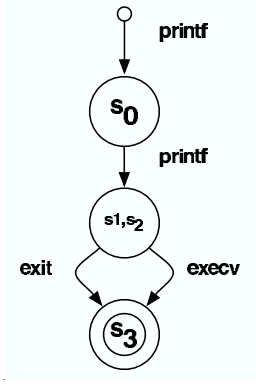

Figure 2.25. DFA model constructed using subset algorithm which is susceptible to a mimicry attack.

Table 2.7
Memory usage in KB as reported by the `pmap` command.

Program	Unmonitored	Monitored	Percentage Increase
htzipd	1,712	2,892	68.92%
lhttpd	2,168	2,580	19.00%
wu-ftpd	3,360	17,364	416.79%
gnatsd	4,772	82,116	1,620.79%

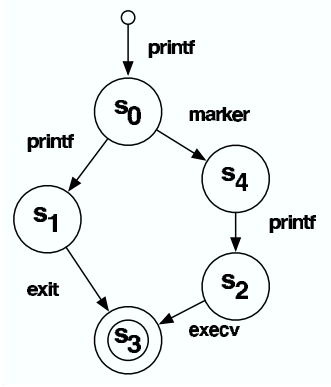

Figure 2.26. DFA model constructed using a marker which is not susceptible to a mimicry attack.

2.7 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 misuse detection. While IAMs obtain run-time performance better than NFAs (which suffer from performance degradation because of increased non-determinism as a result of context-insensitivity), the footprint of an IAM for a typical run[1] can be rather large as shown in Table 2.7. Figure 2.10 presented a compact data layout for the model. Here we study automata compaction techniques.

2.7.1 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.

[1]The `pmap` output in Linux was surprisingly different for different runs of the same program. The numbers reported are for a typical run.

Table 2.8 shows the reduction in the number of states when multiple final states are merged into a single final state in the automaton. Each final state that is 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. The current implementation already takes advantage of this compaction technique.

Table 2.8
Reduction in the number of states when multiple final states are merged into a single final state in the automaton.

Program	Number of States Before Merging	Number of States After Merging	Percentage Reduction
htzipd	24,305	24,273	0.13%
lhttpd	414	401	3.14%
wu-ftpd	170,237	169,763	0.28%
gnatsd	253,230	221,481	12.54%

2.7.2 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 20 bits. Delta successors do not use identifiers for the successors of a state, instead they use offsets (in the `model`) of the successors relative to the current state. Such successor states are *delta successor states*.

When delta successors are used in IAM models, the MDS monitor has to process extra logic (implemented as a `switch` statement) necessary to determine the number of bits used for the delta successors and then calculate the actual offsets of the successors. This adds more overhead as shown in Table 2.9. This is noticeable mostly in Workload-Two for `gnatsd` because the number of library calls intercepted is very high (see Table 2.5) and so the extra logic is executed each time. This effect is not observed in `lhttpd` although the number of library calls intercepted in `lhttpd` is greater than `gnatsd` because the

Table 2.9
Effect of delta successor states on performance overhead.

Program	Unmonitored Time	Monitored Time with Markers	Monitored Time with Markers and DeltaSuccs	Percentage Overhead
htzipd	26.56s	26.95s	26.71s	0.56%
lhttpd	16.16s	16.56s	16.20s	0.25%
wu-ftpd	18.44s	18.28s	18.80s	1.95%
gnatsd-Workload-One	46.91s	47.17s	48.05s	2.43%
gnatsd-Workload-Two	34.58s	45.34s	104.17s	201.24%

number of states in `lhttpd` is significantly lesser than `gnatsd`. This means that the number of bits used in delta successors does not vary as much as it does in `gnatsd` and so the extra logic is not executed to its limit.

Table 2.11 shows the reduction in model sizes that can be obtained by using delta successors. These numbers were calculated by observing the number of bytes and bits we actually need. The memory allocation routines (`malloc` and `calloc`) do not allocate the exact number of bytes requested because they try to prevent fragmentation and maintain word boundaries. To be able to observe these reductions, one would have to implement a memory allocator that simply allocates (say using `calloc`) a block of memory of precomputed size equal to size of the model and then during model generation hands out the exact number of requested bytes from the block. And to obtain the reduction observed by allocating the exact number of bits (instead of the nearest byte), one would have to perform more bit arithmetic and can expect more overhead.

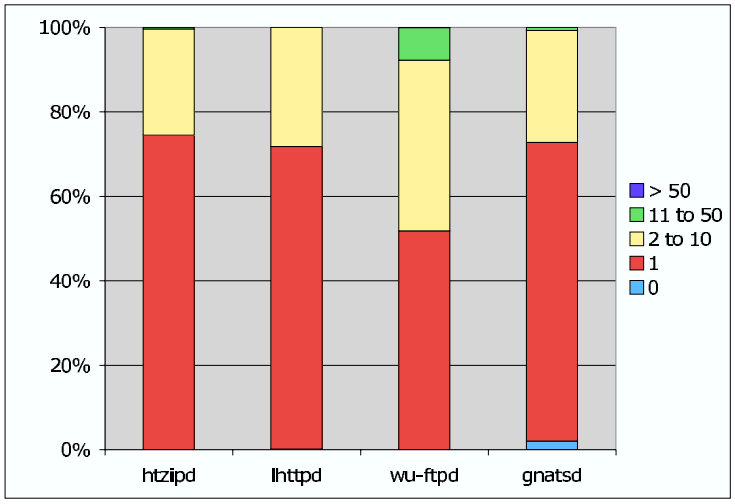

Figure 2.27. Percentage distribution of fan-out values among automaton states before ϵ-reduction.

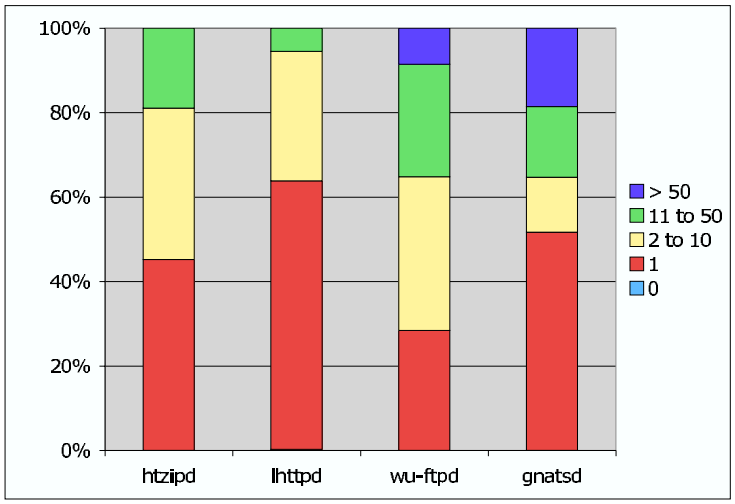

Figure 2.28. Percentage distribution of fan-out values among automaton states after ϵ-reduction.

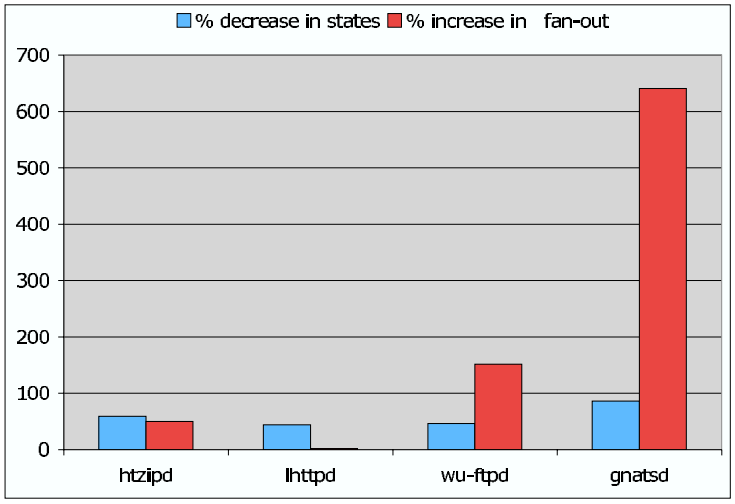

Figure 2.29. Percentage decrease in number of states and percentage increase in fan-out due to ϵ-reduction.

2.7.3 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 misuse 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 fan-out in the model by introducing more transitions per state. Each automaton state now has a greater fan-out than before. Figures 2.27 and 2.28 show the distribution of fan-out values before and after ϵ-reduction. Figure 2.29 shows the percentage decrease in the number of states and the percentage increase in fan-out when ϵ-reduction is performed.

So, including ϵ-transitions in the model actually reduces the model size. The data structure of the ϵ-IAM model is different from the ϵ-free IAM. The model is again represented by a table of nodes. 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) and a `succ` bit to indicate if this is the last node. Each state now has a one bit identifier to distinguish a library state from a non-library state. Each state also has a 11 bit library function identifier `funid` which is zero if the state is a non-library state. Table 2.11 shows that including ϵ-transitions can reduce the `wu-ftpd` model size by as much as 62.91%. But in the case of `lhttpd`, it increases the model size because the additional fields needed in the ϵ-IAM model's data structures offset the decrease in the number of transitions.

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. We can observe from Table 2.10 that including ϵ-transitions adds reasonable overhead except in the case of `gnatsd`. The `gnatsd` models with ϵ-transitions were terminated after about 5 minutes of making no progress. This means that this compaction technique has to be used on a per-application basis.

Table 2.10
Effect of including ϵ-transitions on performance overhead.

Program	Unmonitored Time	Monitored Time	Monitored Time with Epsilon	Percentage Overhead
htzipd	26.56s	26.95s	31.98s	20.41%
lhttpd	16.16s	16.56s	17.06s	5.57%
wu-ftpd	18.44s	18.28s	19s	3.04%
gnatsd-Workload-One	46.91s	47.17s	too slow	–
gnatsd-Workload-Two	34.58s	45.34s	too slow	–

Table 2.11
Model size in KB and percentage reduction on using deltaSuccs and epsilon transitions.

Program	Model	Model with DeltaSuccs	Model with DeltaSuccs (bits)	Model with Epsilon	Model with Epsilon (bits)
htzipd	630.59	9.13%	17.70%	2.30%	13.98%
lhttpd	5.5	16.15%	23.47%	-28.81%	-13.69%
wu-ftpd	12,546.97	11.49%	18.04%	56.97%	62.91%
gnatsd	75,397.63	1.54%	8.67%	–	–

2.7.4 Excluding Functions

Excluding functions from the model can reduce the size of the model. [8, 9] found that ignoring the `brk` system call from the model reduced the size of the model and also the ambiguity in the model. This improved the performance. Others [51] have limited the monitoring to only a small subset of all system calls which they deemed as "dangerous" to reduce the monitoring overhead. Not intercepting certain functions will allow an attacker to invoke those functions without being noticed by the MDS.

We excluded functions on a per-application basis as recommended by [8, 9]. We ignored calls to `getc` and `putc` in all the models. These two functions imposed significant overhead for `wu-ftpd` because they are invoked for every character that is transferred from the server to the client.

For `gnatsd`, we added `malloc`, `free`, and several string functions such as `strlen`, `strcmp`, `strncmp`, and `strchr` to the list of ignored functions. These string functions do not invoke any system calls and are relatively harmless. While `malloc` and `free` could potentially be misused by an attacker for launching denial of service attacks, such attacks are beyond the scope of model-based MDSs [8, 9]. Evaluating the impact of ignoring particular library functions or system calls to the effectiveness of a MDS is an area of future research.

2.7.5 Adding Markers

Section 2.6 described the concept of markers. Adding switch markers unexpectedly reduced the size of the `wu-ftpd` model by 62.28%. This illustrates that such markers can be used as a compaction technique besides being a solution to reducing non-determinism.

2.7.6 Combining Equivalent Transition Symbols

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 [10] we hypothesized that if the number of successors is much greater than the number of unique transition symbols per state then we would save about one byte per transition with a common function symbol. And we observed that this was the case especially for `gnatsd`. However, we realized that such an opportunity also indicates extensive non-determinism which would negatively impact the performance. Because we resolve such non-determinism using deterministic markers, there is less opportunity for using this compaction technique.

2.7.7 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. This compaction technique 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. The space savings come from the fact that a transition can be encoded as a sequence of bytes (about one per symbol) and that the state space is reduced as the nodes for the single-successor state are not needed. We calculated that 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 [10]. And given the distribution of fan-out values for states in our benchmark suite, we hypothesized that there was potential for significant reduction in the number of automaton states using this technique.

However, while designing the data structures for such a model, we realized that this would upset the constraint of the successors having equal sized data structures because the number of coalesced states per successor could vary. This meant that we would have to use a linked list instead of an array of successors, which would in turn add 32 bits (pointer size) for every successor. This would offset the benefits of the compaction. Also the benefits of coalescing single successor states can be fully realized only when the head of the list of single successor states has only one parent or else the coalescing would have to be duplicated at each of the head's parents. For this reason, we did not construct and evaluate a model with this compaction technique.

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.

2.8 Monitoring Interface: Efficiency

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 [9, 41]. Other approaches typically analyze statically-linked binaries [5–7]. The static analysis infrastructures used in our prototypes [44, 45] were simply not able to handle these libraries.

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.

Recently, [52] constructed system call level IAM models for Solaris binaries and reported that inlined models at the system call level are significantly larger than the ones at the library level reported in [10]. The construction of models for `htzipd` and `gnatsd` did not even terminate because of memory overhead. They argue that Lam and Chiueh [41] are able to build inlined models at the system call level possibly because of three reasons. First, they build models for Linux whose library code is less complex than that of Solaris. Second, they resolve function pointer targets at runtime using notify calls without inlining them in the model. And third, they consider statically linked applications thus avoiding the complexity of the runtime linker in the model.

This leads us to infer that system call level inlined models can be used for efficient MDS on Linux systems using our approach combined with the runtime function pointer resolution of [41]. And system call level inlined models will not be feasible for Solaris binaries if they are dynamically linked and if targets of function pointers are inlined in the model. The second concern can be addressed as part of model construction and the runtime monitor as shown in [41] but the first concern related to statically linked applications will have to be evaluated in future work. While statically linked libraries provide a security benefit by preventing attackers from linking to malicious libraries at runtime, they have manageability implications because applications will have to be recompiled and linked when newer versions of libraries become available.

2.9 Monitoring Interface: Effectiveness

This section compares the benefits and drawbacks of the monitoring interfaces with respect to the effectiveness of a MDS. Ideally, an MDS should monitor every single statement executed by the target program and validate each machine instruction. While this is not currently feasible, existing models are approximations at different levels of granularity.

The coarser the approximation, the easier it is to mount a mimicry attack. For example, restricting the observable events to system calls means that library function calls that do not make any system calls are not captured in the model. [5, 7] dynamically prune from the model symbols guarding a function that does not issue a system call. 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 misuse. Or it would have to rely on other specialized techniques to detect such deviant behavior.

Monitoring at the library interface gives us the unique advantage of being able to integrate into our MDS the dynamic detection and avoidance of format string and buffer overflow vulnerabilities caused by library functions. Section 2.10 describes the approach to and advantages of performing dynamic checking of program properties at the library interface in the context of our MDS implementation.

Monitoring the library interface alone is not sufficient for misuse detection. Library interposition techniques, such as the one used here, allow us to monitor the library functions called by a target program. However, an MDS based solely on library interposition will not be effective against so called “code injection attacks” because if an attacker manages to exploit a vulnerability without setting off the MDS then he can further evade the MDS by directly issuing system calls in the “attack code.” However, such a MDS can rely on other specialized techniques that prevent the execution of injected code. These include non-executable stack and heap [53, 54] and randomized instruction sets [55, 56].

The attack model presented in [51] considers attacks that modify program control flow at three levels of granularity: jumping to system call instructions within the injected code (code-injection attacks), jumping to LIBC code (so called “return-to-libc attacks”), and jumping to existing application code. A MDS based on library interposition such as ours is capable of detecting the last two kinds of attacks. Because we model the application

behavior in terms of the library calls, jumping to a library function in the interposed LIBC that is not expected by the model will generate an alert[2]. Similarly, transferring control to a different part of the application code will generate an alert if the library calls issued along that part deviate from the model.

Coupling library interposition with kernel-level system call interposition can leverage the benefits of both techniques. Such a combined MDS 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.

2.10 Dynamically Checking Program Properties using Library Interposition

Our MDS, like other model-based MDSs, dynamically checks if the runtime program control flow deviates from the model generated at compile-time. The same infrastructure can be used to dynamically detect and avoid library function related format string and buffer overflow vulnerabilities using compile-time information to enable and optimize the dynamic checks. We can also dynamically check dataflow intensive program properties for which static analysis alone is either too conservative or unsound.

2.10.1 Format String and Buffer Overflow Vulnerabilities

The `printf` family of C functions such as `printf`, `fprintf`, `sprintf`, `snprintf`, and `syslog` are susceptible to what are known as format string vulnerabilities. This type of vulnerability is an artifact of C's variable-argument mechanism being type unsafe. The `printf` family functions, which use the variable-argument mechanism, determine the number, order, and type of arguments based on the format string specifiers supplied to them as part of the format string argument. Of particular interest is the `%n` specifier which writes the number of bytes output so far by the printing function to the memory location

[2]While it may be possible to bypass the interposed LIBC and return to the original LIBC, thereby evading the MDS altogether, we have not evaluated the feasibility of such an attack either locally or remotely.

supplied as an argument. While all the other format specifiers only result in memory reads, the `%n` specifier causes a memory write. This results in a potential format string vulnerability if the format string argument is not a constant but is user-supplied. This is because the attacker has the freedom to control the number and type of format specifiers and so can pop bytes off the stack till the desired memory location is reached and then write to that location using the `%n` specifier.

There are several static and dynamic techniques to prevent, detect, and avoid format sting vulnerabilities. Static approaches include PScan [13] and GCC's `-Wformat=2` flag that detect non-static (not a constant) format string arguments, percentS [14], which uses static taint analysis to determine user-modifiable format string arguments, and CCured [57] and Cyclone [58], which are type-safe versions of C. Pure dynamic approaches include the library-interposition based `libsafe` [59] that prevents `%n` from writing to a function's return address but does not work with programs compiled with GCC's `-fomit-frame-pointer` flag because it requires frame pointers.

FormatGuard [60], which is available as a modified version of `glibc` (the C library), counts the number of arguments to `printf` family functions at compile-time using macros, uses this count in a `__protected_printf` function, and compares this count to the number of format specifiers in the format string argument at run-time. The number of format specifiers is obtained using the `parse_printf_format` function in `glibc`.

We can implement a FormatGuard-like approach to detect and avoid format string vulnerabilities in the context of our MDS. Using CIL, we can identify the `printf` family functions and count the number of arguments. This count can be embedded in the program text using markers. Every call to `printf` family function would be immediately preceded by a format string marker call that has the `printf` argument count as its only parameter. The interposed libraries for such `printf` functions would extract the count from the format string marker using a global variable. This count can then be compared with the number of format string specifiers obtained from a call to `parse_printf_format`. This approach is as powerful as FormatGuard, does not require the use of a modified `glibc`, and works

in the context of our MDS using library interposition and markers. Furthermore, we can also optimize by limiting the checking to only those `printf` family function calls that use a non-static format string. This can be achieved by selectively inserting markers and resetting the global argument-count variable to a default "dont-care" value after using it in a check. [61], whose approach is more powerful than FormatGuard, reports that such an optimization reduced their performance overhead from 14.1% to 0.7% on `man2html`, a `printf`-intensive application.

C string functions such as `strcpy` and `gets` are susceptible to buffer overflow because of the lack of automatic bounds checking in C. [62] discusses several static and dynamic buffer overflow detection and prevention techniques that have been proposed in the literature. `Libsafe` [63] and `LibsafePlus` [62] are dynamic buffer overflow avoidance approaches that use library interpositioning to intercept the vulnerable functions and perform bounds checking on them. These approaches can be incorporated in our MDS framework with the additional optimization of performing checks only when the source string that is being copied is non-static. This can be achieved by selectively inserting markers to disable checks.

This approach of using selective insertion of markers at compile-time to control the dynamic checks performed in the interposed library can be generalized to a dynamic policy enforcement framework for library functions. For example, a marker could be used to specify the files that could (or should not) be executed by a particular call to an `exec` family function. This would allow a per-callsite policy enforcement instead of a per-application enforcement [64].

2.10.2 Other Properties

MOPS is a model checking tool that detects violations of temporal program properties [12, 19]. In this section, we describe three of the several program properties checked by MOPS. Because MOPS is limited to checking only the control flow aspects of program properties in a sound manner, it ignores the dataflow component of these three properties. This results

in 85% false positives for one of the three properties [19]. MOPS trades precision for soundness and scalability to real-world programs. Other static approaches use dataflow analysis techniques based on unsound heuristics to scale and limit the number of false positives [15]. We use the three properties from MOPS as examples to illustrate that dynamic checking of such dataflow related properties can be easily integrated with our library interposition based approach with zero false positives and zero false negatives.

Property 1: "A process should drop privilege from all its user IDs before calling `execl`, `popen`, `system`, *or any of their relatives."*

This suggests that it is dangerous for a privileged process to invoke functions that can execute untrusted programs. Statically checking this property consists of two parts: determining the location of calls to such functions and determining the privilege level at such program points. The first part can be inferred from the control flow graph of the program except when such calls are made using function pointers which then requires pointer analysis. The second part involves not only tracking the `setuid` family of uid-setting system calls [65] but also determining the value of their arguments because the arguments specify the new `uid`. This second component is dataflow related and even with dataflow analysis it is not always possible to determine the data value from program text unless the data values used are constants or string literals. This means that a sound static analysis tool to be conservative will always have to assume that a setuid call elevates privileges. For this reason, MOPS ends up with a 85% false positive rate for this property [19].

This property can be checked in the interposed versions of the dangerous functions. Before making the call to the actual function, we can accurately (with zero false positives and zero false negatives) determine the privilege of the process that issued this function call using the `getuid` family of function calls. If the process is executing with elevated privileges (as deemed by a policy) then we can generate an alert, prevent the execution of that dangerous function, or even abort that process.

Property 2: "The privilege of a process when it calls `longjmp` *must match its privilege when it calls* `setjmp`."

Property two suggests that the privilege level of a process at the time of a `longjmp` call should match the level at the time of the matching `setjmp` call issued earlier. Besides the static privilege determination that was needed for the first property, this property also requires the determination of matching `setjmp` and `longjmp` calls. This matching requires dataflow analysis to establish that the `env` buffer used by the calls is the same. This introduces more conservativeness and therefore the likelihood of a greater false positive rate than present in property one. Instead, we can perform dynamic checking in the interposed version of `longjmp`. The interposed version of `setjmp` simply records the `env` buffer and the current privilege level. In the `longjmp` call, the privilege level of the corresponding `setjmp` call can be accurately retrieved by matching the `env` buffer and this can be checked against the current privilege level.

Property 3: "The `setuid` *system call should never fail."*

This is a specific case of a general property that suggests that return values of library functions and system calls such as `malloc` and `setuid` should be checked for failure. Not doing so might result in a vulnerability. Sound static checking of this property has to rely on pointer analysis and dataflow analysis because the return values may flow to other data values before being checked. Such checks can be automatically and accurately done in the interposed versions of necessary functions at runtime and alerts can be logged.

These examples illustrate that the library interposition infrastructure can be used to perform pure dynamic checks for dataflow-related properties for which current static analysis tools are either overly conservative or unsound.

2.11 Limitations and Future Work

The existing approaches to anomaly detection primarily address only a part of the MDS problem: accurate and efficient monitoring of function call sequences. This is the simplest, yet important, concern for misuse detection because an attacker has to use system calls, or library functions that serve as system call wrappers, to interact with the under-

lying operating system to cause harm (with the possible exception of DoS attacks). By accurately modeling the acceptable sequences of function calls, the models limit the attacker to only those expected call sequences. However, there are several limitations that diminish the precision of these models.

1. 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 introduces impossible paths that can be exploited by an attacker as illustrated in [5].

2. Data flow Analysis.

   Data flow support is another requirement for more robust MDSs. It is well documented [7, 9] 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 MDS 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. We focus on lightweight control-flow analysis in our research. Recent work has considered context-sensitive dataflow [66] and dataflow anomaly detection [67].

3. Object-Oriented Languages

   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 [68], *i.e.* it is possible to determine unambiguously which implementation will be invoked.

2.12 Related Work

In this section, we discuss research related to MDS but not specifically to context-sensitive model-based MDS. We have discussed that in Section 2.1.

Giffin et al. [66] constrain the model at program load time by considering the execution environment. This includes information from configuration files, command line, and environment variables. They show that this improves precision of models at the expense of a one-time increased overhead at program load time.

Xu et al. [51] propose an approach based on adding "waypoints" to monitor program control flow and restrict permissions. They add waypoints at the entry and exit of every application function. These waypoints encode permissions in the form of system calls that are allowed to be invoked from the function being guarded. At runtime, a stack of waypoints determines the permission context. They optimize by adding waypoints to only those functions that make a system call from their list of dangerous system calls after observing that the general case incurs a significant performance overhead. They claim that their approach can support a variety of model-based context-sensitive MDSs because the waypoints provide contextual information [69].

Abadi et al. [70] consider the broad problem of enforcing control-flow integrity in the context of applications such as misuse detection, inlined reference monitors (IRM) [71], and software fault isolation [72]. The legal control flow is embedded in the code using unique bit patterns termed as IDs and the ID checks are enforced at runtime using specialized machine instruction sequences that effectively inline the enforcement of checks. They present a formal treatment of control-flow integrity and evaluate an implementation for the Windows/x86 platform based on binary rewriting.

Injected code can also be prevented from executing using several approaches including software solutions for making data memory non-executable [53, 54], native hardware support for enforcing non-executable data [73], and randomization techniques [55, 56].

3 FAULTMINER: DISCOVERING UNKNOWN SOFTWARE DEFECTS USING STATIC ANALYSIS AND DATA MINING

Program verification techniques can be used to improve the quality of software, and, as a side effect, its resilience to security breaches. Given a specification of correct program behavior, it is often possible to check statically that invariants hold on all possible execution paths. Unfortunately, manually specifying program invariants has proven to be difficult for practitioners. In the absence of program-specific invariants, we are limited to checking generic properties that pertain to known vulnerabilities of the programming language, libraries, or operating system.

There are many security-relevant known program properties that rely on the temporal ordering of program events. Consider, for example, the following temporal properties (where $\rightarrow$ denotes a happens-before relationship): [`isnull(ptr)` $\rightarrow$ `*ptr`]. The fact that it is a responsibility of the program to check that a pointer is non-null before access is a property of the programming language (in Java, for instance, null checks are performed by the virtual machine and the program is only expected to catch any exceptions resulting from the check). The synchronization primitives `lock` and `unlock` should strictly alternate along all paths; thus we must ensure that [`lock` $\rightarrow$ `unlock`]. Forgetting to unlock after locking, double locking, and double unlocking are security violations. As a last example, take the `chroot` function that changes the root directory to be its argument. This is used to confine a process to the portion of the filesystem denoted by the new root. The correct way to create a `chroot` "jail" is to call `chdir("/")` after the call to `chroot` thus changing the current directory to the new root and preventing subsequent attempts to follow upward references ("`..`"). Therefore the property to be checked is [`chroot` $\rightarrow$ `chdir("/")`].

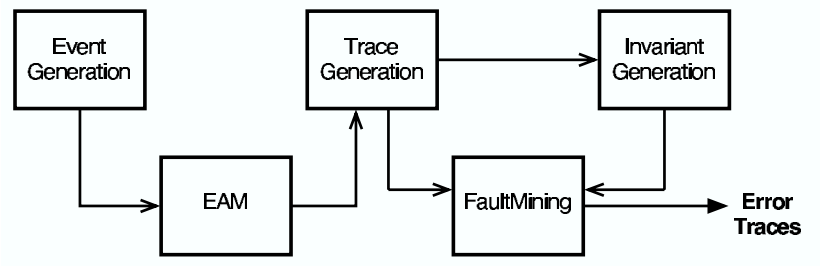

Figure 3.1. FaultMiner framework.

However, programs have many more invariants that are specific to the application logic. These go beyond the simple language- and operating-system-specific properties illustrated above. These invariants are just as critical for security but are unfortunately almost never properly documented. The challenge addressed by the approach described in this research is to find automated techniques for extracting program invariants from the source code with limited user interaction. While we focus on security properties, the approach is clearly applicable to any software defect. We propose a new approach, and a tool named FaultMiner, based on a combination of static program analysis and data-mining techniques for discovering likely invariants. These invariants are used to find software defects that are then presented to the developer or code auditor for reviewing.

We follow the premise of previous work on inferring invariants: common behavior is often correct behavior. This is not necessarily the case, of course, because what a tool may infer as invariants might simply be coincidences or in the worse case, they could be incorrect: code segments reproduced several times e.g. a cut-and-paste error. For this reason, the inferred invariants are usually referred to as *likely program invariants*. We call defects resulting from a violation of such likely invariants as *unknown defects* because the invariants are not known a priori. Recently, several approaches have been proposed to infer likely invariants from a program with the goal of finding defects resulting from a violation of the inferred invariants. They broadly fall into two categories: *dynamic* approaches [22–

25], which observe a program's runtime behavior, and *static* approaches [27–29], which analyze program text to detect likely invariants.

As an example of such invariants, consider the `openssh` program. The function `packet_start` has to be called before `packet_send`, because the former initializes packet construction by appending the packet type. This invariant can be inferred by observing that the sequence [`packet_start` → `packet_send`] occurs 39 times in the `sshd` code. Another sequence, [`buffer_init` → `mm_request_send` → `mm_request_receive_expect` → `buffer_free`] occurs 12 times. It so happens that forgetting any one of the calls in this sequence will be erroneous. Deriving and checking invariants at the level of user-defined functions enables us to detect defects at a higher level of abstraction.

Static approaches are appealing because they have the advantage of observing all the paths in a program. The current static approaches to finding unknown defects consider simple temporal invariants, in specific contexts, and use ad-hoc techniques. For example, Engler et al. [27] and Weimer and Necula [28] consider only function pairs in their invariants. Li and Zhou [29] ignore control flow from conditional statements and consider the function body as a single path. FaultMiner overcomes these limitations.

In this research, we propose a general approach to finding unknown defects. An overview of the FaultMiner framework is illustrated in Figure 3.1. We consider temporal invariants on general *events* (including assertions on data values) that are abstracted using static analysis in an *Event Automaton Model* (EAM). Event traces generated from the EAM are used to infer likely invariants. The FaultMiner analyzes the event traces and the likely invariants to generate error traces.

The technique of inferring likely invariants and finding unknown defects is derived from well-known data-mining algorithms. We describe how two security-critical program invariants can be derived using this novel technique. We present experimental results for FaultMiner on the latest versions of `wu-ftpd`, `cups`, `openssl`, and `openssh`. These are four extensively-used security-critical real-world programs. Using FaultMiner, we

found two new potential vulnerabilities (one in `wu-ftpd` and one in `cups`) and four previously known bugs (in `openssh`), and several other violations.

The rest of the chapter is organized as follows. Section 3.1 describes event generation, EAM, and trace generation. Section 3.2 explains invariant generation. Section 3.3 describes the FaultMining technique and the two security properties. Extensive experimental evaluation is presented in Section 3.4. Section 3.5 discusses the concept of obligations and how FaultMiner can be used to identify pending obligations and also to minimize obligations. Sections 3.6 and 3.7 discuss the challenges that need to be overcome in future and related work respectively.

3.1 Event Automaton Model

```
main() {
  if (..)
    foo("Secure");
  else
    bar();
}

void foo(char *str) {
  write(1,str,strlen(str));
}

void bar() {
  foo("Insecure");
}
```

Figure 3.2. An example program.

Fault mining can be performed on any program representation. It only requires some notion of interesting program events and a partial order among these events. Different static analysis techniques can be used to generate events that can be the basis for invariant inference. In FaultMiner, we decouple the stages of event generation and invariant inference by abstracting the program in an Event Automaton Model (EAM) and inferring invariants on the event traces generated from the EAM.

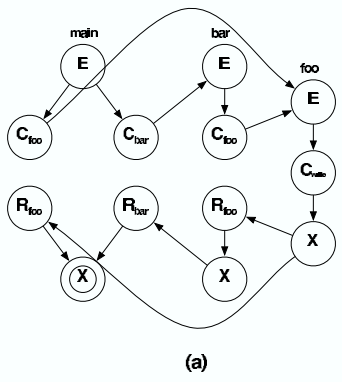

Figure 3.3. ICFG representation of the program in Figure 3.2. E, X, C, and R represent the entry, exit, call, and return nodes respectively.

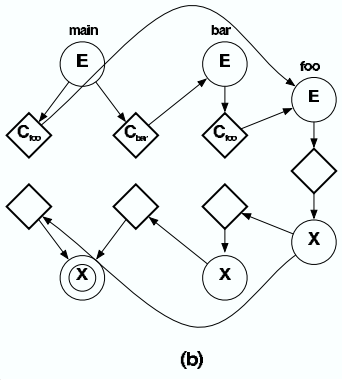

Figure 3.4. EAM representation of the program in Figure 3.2 for user-defined function invocations. E, X, and C represent the entry, exit, and call nodes respectively.

For our purposes, an EAM is a Non-deterministic Finite Automaton (NFA) constructed out of the program's interprocedural control flow graph (ICFG), i.e. the union of statement-level control flow graphs for all functions, and an *event filter*. The ICFG is straightforward, 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 event filter is a function that maps basic blocks to set of events. An event can be

any predicate that holds at a given program point. A program point may generate zero or more events. Examples of events include function calls, def/use of a variable, etc. Events can be of arbitrary granularity. As an example, events of interest for static taint analysis are of the form `is-tainted(x)`.

Event traces can be generated from the EAM by considering sequences of events along all the paths in the EAM. Each trace corresponds to events generated along one particular execution of the program. This can be done in a flow- and context-sensitive manner, e.g. using a pushdown automaton to match the call and return of functions. Trace generation can also be path-sensitive provided we have that information from analyzing the predicates of conditionals. If not, there will be, as usual, infeasible traces caused by path-insensitivity.

An example program, its ICFG representation, and the corresponding EAM where the only events of interest are calls to the two user functions are shown in Figure 3.2, Figure 3.3, and Figure 3.4 respectively. Trace generation for this EAM produces two traces [`foo`] and [`bar` $\rightarrow$ `foo`]. Practical considerations with trace generation in FaultMiner are discussed in Section 3.4.1.

3.2 Mining Likely Temporal Invariants

Informally, temporal invariants are discovered by identifying event patterns common to multiple event traces. The remaining traces where the common event patterns are absent are likely error traces. The challenge is in figuring out what patterns to consider and then efficiently searching for these patterns in event traces. Efficiency is an important factor because of the exponential number of event traces and the complexity of finding all possible common patterns of all lengths across all the event traces. There has been significant research in the area of data mining that has investigated efficient algorithms to discover complex data-relationships in large databases. Our FaultMiner algorithm is based on the work of Agrawal and Srikant [74]. The terminology used in the FaultMiner algorithm is defined below.

Definition 3.2.1 *An event sequence σ is a string $\langle e_1, e_2, \ldots, e_n \rangle$ where e_j occurs after e_i if $j > i$. The length of an event sequence is the number of events present in the sequence. An event sequence of length k is called a k-sequence.*

For a given EAM, an *event trace* is a string in the language represented by the EAM. The set of all the strings in the language is represented by *E*. For clarity, we will speak of event sequences only when referring to properties of interest.

Definition 3.2.2 *A subsequence σ' of a sequence σ is a new sequence derived from σ by deleting one or more of its elements without disturbing the relative positions of the remaining elements. We use the notation $\sigma' \sqsubset \sigma$ to indicate the subsequence relationship.*

Definition 3.2.3 *An event trace τ supports an event sequence σ if $\sigma \sqsubset \tau$. The support S of an event sequence σ for a set of traces T, is the percentage of event traces in T that support σ.*

Definition 3.2.4 *An event sequence with support greater than user-defined minimum support value, minsup, is called a large event sequence.*

Definition 3.2.5 *We define the confidence C of an event sequence σ w.r.t a subsequence σ' as the percentage of event traces containing σ' that also contain σ. This can also be defined as $C^{\sigma}_{\sigma'} = \frac{support(\sigma)}{support(\sigma')}$.*

We consider two user-defined confidences *lowconf* and *highconf* such that $0 <$*lowconf* $<$ *highconf* < 1.

Algorithm. The FaultMiner AprioriAll algorithm is used to generate the set of all large k-sequences of events. These large event sequences represent "common behaviors" or likely invariants because, by definition, they are present in a majority of event traces, where majority is characterized by the value of *minsup*. The algorithm makes multiple

```
algorithm: AprioriAll
input:  Set of event traces E
output: Set of large sequences S
 L_1 ← large 1-sequences
 k ← 2
while L_{k-1} ≠ 0 do
    C_k ← apriori-generate L_{k-1}
    foreach t ∈ E and c ∈ C_k do
      if c ⊏ t then count_c ← count_c + 1
    foreach c ∈ C_k do
      if support(c) ≥ minsup
      then L_k ← L_k ∪ c
S ← S ∪ L_k
k ← k + 1
```

Figure 3.5. The FaultMiner AprioriAll algorithm. L_k represents the set of of all large k-sequences. C_k represents the set of candidate k-sequences.

```
algorithm: apriori-generate
input:  Set of large sequences L_{k-1}
output: Set of candidate sequences C_k

 // Join Phase
 foreach l, l' ∈ L_{k-1} s.t. l ≠ l' do
   let l = ⟨e_1...e_{k-1}⟩ and l' = ⟨e'_1...e'_{k-1}⟩
   if e_1 = e'_1 ... e_{k-2} = e'_{k-2} then
     C_k ← C_k ∪ ⟨e_1...e_{k-2},e_{k-1},e'_{k-1}⟩

 // Prune Phase
 foreach c ∈ C_k and c_s ⊏ c do
     if c_s ∉ L_{k-1} then C_k ← C_k \ c
```

Figure 3.6. The FaultMiner Apriori-Generate algorithm. L_k represents the set of of all large k-sequences. C_k represents the set of candidate k-sequences.

passes over a set of event traces. In each pass, the large sequences from the previous pass are used to generate candidate sequences using the apriori-generate function. The support for candidate sequences is calculated to determine the new set of large sequences. The algorithm is seeded with the set of large 1-sequences.

The apriori-generate function yields candidate k-sequences from large k-1 sequences by first joining the large k-1 sequences and then pruning those candidates that contain any k-1 subsequence that is not large. The pruning phase is the key idea of the AprioriAll algorithm. The underlying intuition is that any subsequence of a large sequence must also be large. This drastically reduces the number of candidate sequences. Agrawal and Srikant [75] provide a proof of correctness of this candidate generation algorithm. The AprioriAll and apriori-generate algorithms are presented in Figure 3.5.

Mining with Constraints While considering event patterns, it is sometimes interesting to consider patterns with certain constraints on event attributes. For example, we might want to consider events that operate on the same memory location, occur in the same context and path, and that have their e_{type} alternating between $type_1$ and $type_2$ (such as calls to `lock(v)` and `unlock(v)`). We support mining event patterns with such constraints by extending the apriori-generate algorithm to apply the constraints in the join phase. So the candidate sequences selected satisfy the constraints in every iteration of the algorithm. This incremental approach to constraint satisfaction prunes the search space and improves efficiency.

3.3 FaultMining

FaultMining is based on the two concepts of sufficient evidence of common behavior and sufficient evidence of deviant behavior. A program path or an event trace is the unit of evidence in FaultMiner. Sufficient evidence of common behavior is captured in the form of likely invariants. Deviant behavior is behavior that deviates from the common behavior. Evidence for such behavior should be enough to classify it as deviant and not great enough

```
algorithm: Maximal-Sequences
input:   Set of large sequences S
output: Set of maximal sequences M

  M ← S
  k ← Length of the longest
        sequence in S
  while k > 1 do
   foreach k-sequence s ∈ M do
     foreach c ⊏ s do
       M ← M \ c
   k ← k – 1
```

Figure 3.7. Maximal-Sequences algorithm.

```
algorithm: FaultMiner
input:   Set of maximal sequences M
           and event traces E
output: Set of error event traces ξ

  ξ ← 0
  foreach m ∈ M and s ⊏ m do
  // Subsequences are chosen in the
  // decreasing order of their length
      if C^m_s = 1 then continue
      if C^m_s > highconf then
        foreach t ∈ E do
          if s ⊏ t and m ⋢ t then
            ξ ← ξ ∪ t
      if C^m_s < lowconf then
        foreach t ∈ E do
          if m ⊏ t and then
            ξ ← ξ ∪ t
```

Figure 3.8. FaultMiner algorithm.

to classify it as another common behavior. This section explains these concepts, describes the FaultMiner algorithm, and illustrates two security properties that can be captured using this technique.

The AprioriAll algorithm generates the set of all large sequences. The Maximal-Sequences algorithm shown in Figure 3.8 takes this set and removes subsequences to retain only the longest sequences. Formally, what this means is that we are able to identify the *complete likely invariants* (CLI) and discard all the *partial likely invariants* (PLI) (subsequences of CLIs) from S.

Definition 3.3.1 *A complete likely invariant (CLI) is a sequence $\chi \in$ set of large sequences S generated by AprioriAll, such that there does not exist any other sequence $\rho \in S$ that satisfies $\chi \sqsubset \rho$. A partial likely invariant (PLI) is a sequence $\phi \in S$ such that $\exists$ χ ($\chi \neq \phi$) that satisfies $\phi \sqsubset \chi$.*

This is a significant improvement over related approaches [27, 28] because PLIs might not be meaningful when presented to a software developer or code auditor using such a tool. Also, multiple PLIs that constitute the CLI will result in redundant checking and redundant alerts.

Our FaultMiner algorithm is illustrated in Figure 3.8. It takes as input a set of complete likely invariants (maximal sequences) M computed by the Maximal-Sequences algorithm and the set of event traces E. For each sequence in M, the algorithm computes the sequence's confidence relative to each of its subsequences i.e. $C_{cli'}^{cli}$. We use the notation CLI′ to denote any subsequence of CLI hereafter. If this confidence is 1 then it means that all the event traces that satisfy CLI′ also satisfy the CLI. This is not of much interest to us for the goal of fault finding. But if this is not the case and if the confidence is higher than the *highconf* then it means that there are a few event traces that satisfy the CLI′ but not the CLI. These are potential error traces—traces with *likely omitted event(s)*. For example, if a call to function `a` is followed by a call to function `b` along 99 paths and there is only 1 path where `a` is not followed by `b`, then it is likely that the call to `b` was omitted by

mistake. Note of course that this exceptional behavior might be correct, thus developer intervention is needed to determine if it is a defect.

On the other end of the spectrum, if the confidence is lower than the *lowconf* then it means that although the CLI was "common enough" to be classified as an invariant, the events differentiating it from the CLI′ may actually be erroneous occurrences. Event traces that satisfy such CLIs are potential error traces—traces with *likely inserted event(s)*. For example, if a call to function `a` is followed by a call to `b` along 1000 paths and there are only 10 of them where there is a call to `c` in-between the calls to `a` and `b`, then it is possible the calls to `c` are errors.

Previous static approaches [27, 28] consider the simplest case of temporal ordering of two events and use ad-hoc techniques to limit the search space. Li and Zhou [29] consider an arbitrary number of events but ignore control flow by treating the entire function body as a single path. Our FaultMiner algorithm is the first generalized control-flow-sensitive algorithm capable of analyzing temporal ordering of an arbitrary number and type of events. This enables us to infer complete likely invariants instead of multiple partial ones along program execution paths.

3.3.1 Security Properties

There are many program properties that are specific to a program's logic and that are not explicitly documented, not well-understood, and whose violations are not caught by most existing program analysis tools. These properties are typically part of non-functional requirements and are expected to be implicitly satisfied by the program. Unknown defects resulting from a violation of such implicit invariants are harder to detect because one has to first infer the implicit invariants before checking if the inferred likely invariants hold along all program paths. Two categories of such unknown defects are described below.

Function Call Sequence (`f` → `g` → ...). Unlike library functions and system calls whose semantics are documented (in the form of `man` pages) and relatively well-understood,

the semantics of user-defined functions or APIs are usually not explicitly documented. So if such functions have to obey some temporal constraints, it is likely that these invariants are known only to the software developer(s). For example, in `openssh`, the function `packet_start` must always be called before calling `packet_send`. This invariant cannot be captured at the level of library functions or system calls. Checking invariants at the abstraction of user-defined functions enables us to detect defects beyond those that manifest from an incorrect usage of system calls or library functions.

One might argue that such invariants are not relevant to security and that their violations will "only" lead to incorrect but benign behavior. This is not true. Identifying incorrect behavior of security-critical programs such as `openssh` and `openssl` is of extreme importance. Often, incorrect behavior can manifest into malicious behavior with the attacker compromising the confidentiality, integrity, and/or availability of the system.

Additionally, the higher level of abstraction of user-functions enables defect detection at a coarser granularity. For example, in `CUPS`, the Common Unix Printing System, the sequence [`cupsFileOpen` → `cupsFileGets` → `cupsFileClose`] occurs 9 times. Each of these functions encapsulates several checks and actions. The `cupsFileClose` function frees the memory associated with the `CUPS` file besides closing the file. In this case, instead of separately checking for memory leaks and file descriptor leaks by analyzing at the level of library functions, it is more efficient to match calls to `cupsFileOpen` with `cupsFileClose`.

Developing software in collaborative environments, with complex requirements, minimal documentation of such implicit rules, and few tools to infer and check these rules is a challenging task. Previous work [27, 28] has looked at techniques to infer temporal ordering between function pairs. While function pairs might be CLIs in a few cases, they may be PLIs in others. Checking for PLIs will result in an exponential blow-up and will also produce less meaningful alerts. Besides, the techniques used do not generalize to sequences of longer length. Recent work by Li and Zhou [29] has attempted to address this concern by proposing a technique to infer temporal invariants among an arbitrary number

of function calls. While this is a definite improvement over the others, the limitation is that it ignores control flow and considers a function body as a straight line sequence of instructions. So if an invariant is satisfied along any one path in a function, it is assumed to be satisfied along all paths. It is well-known that security violations usually occur along exceptional paths or paths that rarely occur at runtime and hence are difficult to detect using conventional testing. Ignoring control flow will therefore miss out on an important property that contributes to security violations. FaultMiner not only generalizes to arbitrary number of function calls but also considers control flow because EAM is obtained from the ICFG.

Invoke → Check-Return → Use-Return. In March 2004, a critical security vulnerability was found in the Linux kernel memory management code inside the `mremap` system call because of a failure to check the return value of a function invoked in the system call code [76]. Checking the return values of library functions and system calls such as `malloc` and `setuid` has long been recognized as a good programming practice. Failure to do so may not always lead to a vulnerability. But it might, when an exceptional condition occurs (such as the function fails and returns an error code) as in the case of the Linux vulnerability.

The return values of user-defined functions should also be checked. Functions that return a non-`void` value might not always return error codes and so their return values may be assigned to variables and used later without performing any checks. Sometimes their return values might not be assigned to any variable and this may be fine according to the program logic. So assuming that all non-`void` returning function calls should be checked is overly conservative. But if there is any instance in the program where a function call's return value is checked before use then this may be evidence that the function returns a value that needs to be checked before using it. FaultMiner identifies such functions using any available evidence and detects violations if return values of such functions are used without being checked.

Table 3.1
Characteristics of evaluated software.

Software	Version	#C files	#Functions	LOC	Description
wu-ftpd	2.6.2	51	221	26,317	A widely-used ftp daemon
cups	1.2	156	308	132,002	Common UNIX Printing System
openssl	0.9.8	767	2,274	259,611	A library of cryptographic primitives
openssh	4.2p1	160	861	66,813	A free version of the `ssh` suite of network connectivity tools

3.4 Evaluation

We have implemented our FaultMiner tool using CIL [45]. CIL (**C I**ntermediate **L**anguage) is a high-level representation along with a suite of tools that facilitates whole program analysis of C programs. FaultMiner is implemented as a CIL module and can be invoked with a command-line argument to CIL driver.

We evaluate the FaultMining concept for the two categories of unknown defects on four widely-used real-world programs. The characteristics of these four programs are shown in Table 3.1. The experiments were run on a 2.8 GHz Linux machine with 1 GB RAM.

3.4.1 Practical Considerations

Trace Generation. Generating traces along paths is an exponential task. For this reason, all approaches, including ours, consider only local paths and ignore interprocedural paths. Although Li and Zhou [29] analyze up to a call depth of three, they avoid the problem of exponential paths by treating the entire function body as a single path because they ignore control-flow. In our work, we solve the problem using several techniques.

We consider local EAMs generated from CFGs and generate event traces from them. FaultMining is performed on local event traces. Currently, we generate a maximum

of 10000 traces per function or generate traces for one minute per function, whichever threshold is reached earlier. We also consider a maximum of 10 events per trace (8 for `openssh`). These values were chosen to attain reasonable memory overheads. To compensate for local paths, we consider non-local evidence in two ways.

1. Non-local Evidence (NLE). First, for every local violation in function F_i of the form [*highconf* $< C^{cli}_{cli'} < 1$], we compute $C^{cli}_{cli'}$ for all the other functions F_j in which both cli and $cli' \in$ large sequences of F_j. We refer to these confidence values as *non-local-evidence* (NLE) because they present statistical evidence of the violated CLI in other functions. We rank the local violations by their average NLE given by $(\sum_{j=1}^{n} F_j(C^{cli}_{cli'}))/n$. For example, if a local violation has an average NLE of 1 over *n* functions, then it means that in *n* other functions, every trace that contains CLI′ also contains the CLI (confidence of 1 or 100%). This makes the local violation a serious one especially if *n* is large. We also rank violations based on whether trace generation for that function was completed or terminated from exceeding the 10000 traces or one minute threshold. A violation is ranked higher if trace generation was completed for its function.

2. Binary Violation. Second, we also detect violations where, for example, a CLI [$F_a \rightarrow F_b$] is present in, for example, 10 functions and there is only one function that has only the CLI′ F_a but not the CLI. We consider it a violation if a CLI is present in more than *binary-support* number of functions, say *n*, and if $n/(n+m) >$ *highconf* where *m* is the number of functions that contain only the CLI′ and is greater than zero. We call such violations *binary-violations* because the violation depends on the presence or absence of the CLI in entire functions instead of a few paths in a function. Binary violations are different from the other violations because the number of paths do not play any role in them. The CLI is completely absent in the violating function.

Data-structure for Support Evaluation. The support calculation for candidate sequences has to be performed on an exponential number of event sequences in the AprioriAll algo-

rithm. This is an expensive operation and needs to be performed efficiently. For this reason, a *hash-tree* data-structure is used [74, 77].

Candidate sequences are stored in a hash-tree. A hash-tree is a tree whose interior nodes are hash tables and leaf nodes are lists of items or in our case, candidate event sequences. Each hash table bucket in the interior node points to another interior node or a leaf node. An implementation of a hash-tree is based on two parameters: *branching-factor*, that specifies the number of buckets in the interior nodes, and the *leaf-threshold*, that specifies the maximum number of event sequences in the leaf nodes. We use a branching-factor of 10 and a leaf-threshold of 100 in the current FaultMiner prototype. Because of space constraints, we refer the interested reader to [74, 77] for a description of the algorithms for hash-tree insertion and support calculation.

Table 3.2
Violations detected for Property One.

Software Evaluated	Events	Traces	Invariants Inferred	% Invariants as function pairs	Violations Detected	Violations with Avg. NLE = 1	Binary Violations	Time
wu-ftpd	971	433,968	4,104	12	2,668	2	7	4m33s
cups	885	361,439	8,470	8	3,403	6	304	5m20s
openssl	4,098	1,076,248	21,459	11	13,930	47	92	23m43s
openssh	5,521	657,400	83,765	7	33,212	348	576	17m52s

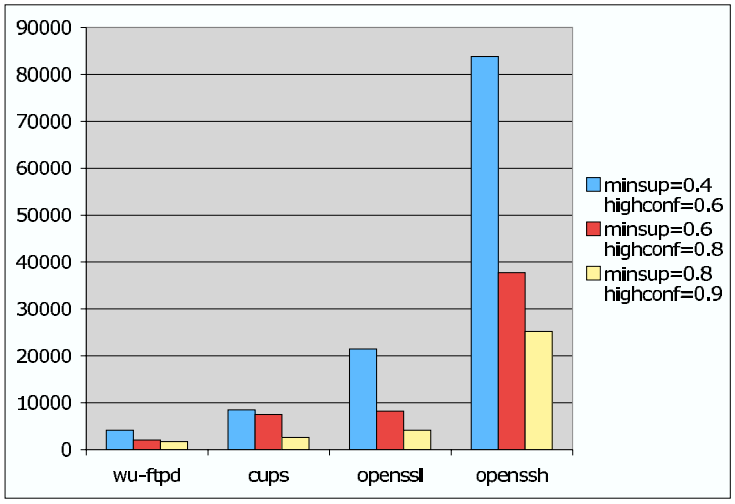

Figure 3.9. Number of invariants generated for Property One at three levels of support and confidence.

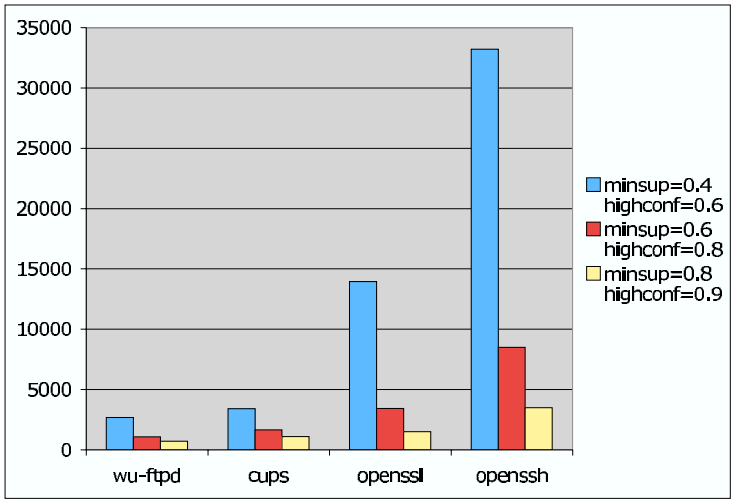

Figure 3.10. Number of violations generated for Property One at three levels of support and confidence.

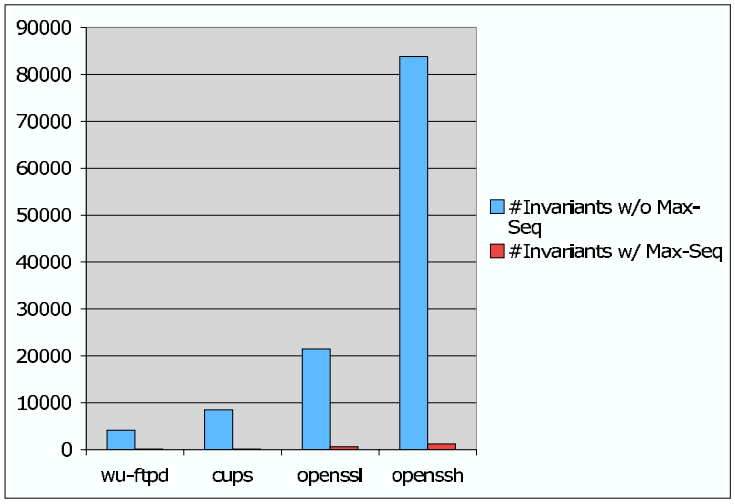

Figure 3.11. Number of invariants generated for Property One without and with running the Maximal-Sequences algorithm.

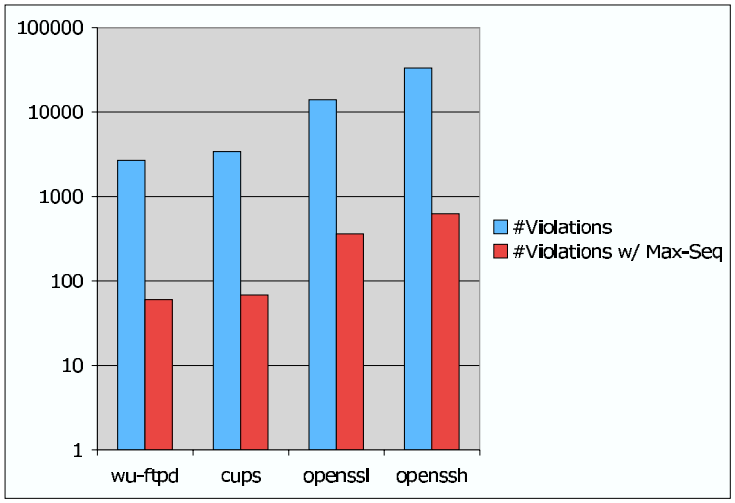

Figure 3.12. Number of violations generated for Property One without and with running the Maximal-Sequences algorithm.

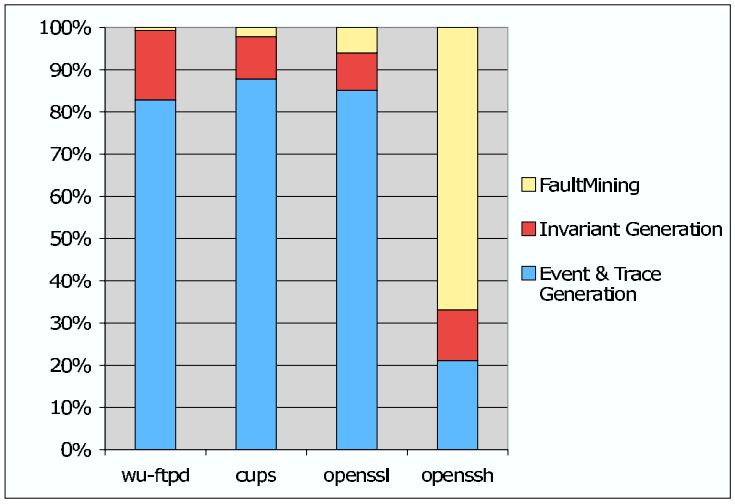

Figure 3.13. Percentage distribution of time among the different stages of FaultMiner for Property One.

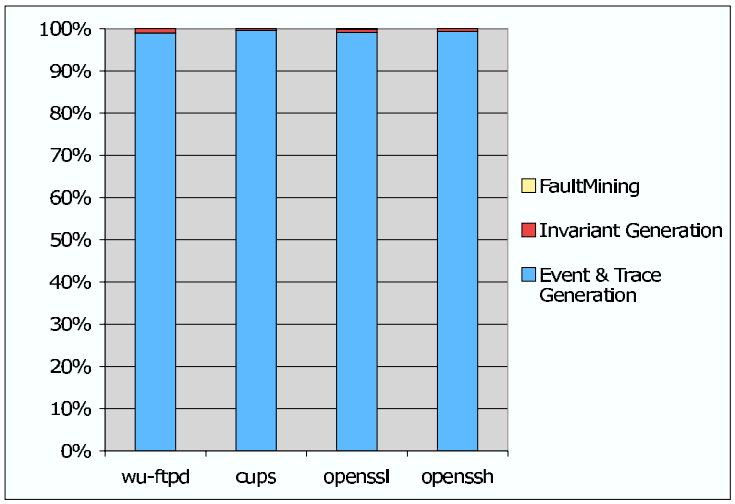

Figure 3.14. Percentage distribution of time among the different stages of FaultMiner for Property Two.

3.4.2 Property One: Function Call Sequences

For this property, calls to user-defined functions are considered as events in the EAM. FaultMining is performed on local event traces. We observed that by running the Maximal-Sequences procedure, we were inferring CLIs that were present in few or no other functions. This was because, one or more function calls would invariably appear on a majority (greater than *minsup*) of the paths following the actual CLI and would be appended to the actual CLI resulting in a coincidental CLI that was present in few or no other functions. For example, if $[F_a \rightarrow F_b]$ were the actual invariant and it was followed by F_c in F_{foo} and by F_d in F_{bar}, along a majority of the paths, then the inferred CLIs $[F_a \rightarrow F_b \rightarrow F_c]$ and $[F_a \rightarrow F_b \rightarrow F_d]$ would not serve as NLE for each other in case of a violation. So we do not run the Maximal-Sequences procedure for this property and instead consider all the large sequences generated by the AprioriAll algorithm in the FaultMining procedure (i.e. $M = S$). Ranking based on NLE ensures that the large sequences that have a greater likelihood of being CLIs are ranked higher than their subsequences or larger coincidental CLIs. Also, we consider only subsequences of length $k-1$ while detecting violations for a large k-sequence and we do not currently detect likely inserted events violations.

Table 3.2[†] shows the number of events, traces, inferred invariants, detected violations, and time taken when FaultMiner was run with a *minsup* of 0.4, a *highconf* of 0.6, and *binary-support* of 2. These are moderate values of support and confidence. Less conservative values will reduce the number of violations but might miss some interesting violations. Figures 3.9 and 3.10 show the number of invariants inferred and violations detected for this property at three different levels of support and confidence. The number of inferred invariants that were function pairs is only about 10% of all the invariants as shown in Table 3.2. The rest of the invariants would have been missed by related approaches that consider only pairs of functions.

The lower number of inferred invariants and detected violations when Maximal-Sequences is used in FaultMiner is illustrated in Figures 3.11 and 3.12 (note that the y-axis is in loga-

[†]Numbers reported for `openssh` are for the `sshd` component

rithmic scale). Figure 3.13 shows the percentage distribution of time spent by FaultMiner in the different stages. Except for `openssh`, trace generation (event generation contributed very little) accounts for most of the time spent by FaultMiner. The surprisingly high number of invariants inferred in `openssh` (possibly because of relatively excessive straight-line code that supports several large sequences) leads to FaultMining time (the binary-violation part of this phase) dominating the other phases. We manually examined all the violations that had an average NLE of 1.0 and also those that had a majority of NLE equal to 1.0. We discuss some of the interesting violations for property one detected by FaultMiner.

wu-ftpd. In function `CheckSum`, FaultMiner detected that while a call to `ftpd_popen` was present on 8 paths, it was followed by `ftpd_pclose` on only 6 paths and this violation had NLE of (1.0, 1.0, 0.0) in the functions `site_exec`, `statfilecmd`, and `retrieve`. On examining the code, we found that the 2 paths where `ftpd_popen` is not followed by `ftpd_pclose` in function `CheckSum` are on the error paths when `ftpd_popen` returns NULL. This suggests that the programmer believes that `ftpd_popen` may return NULL and that its return value needs to be checked. And indeed, when we examine the function `ftpd_popen`, there are six places where it may return NULL. Five of them are on failures of the library functions `getrlimit`, `getdtablesize`, `calloc`, `pipe`, and `fdopen`. These functions could fail when the accessed resources are exhausted. This means that if the return value of `ftpd_popen` is used without checking for NULL, then upon a resource-exhaustion attack, the server will crash instead of a graceful failure. With this analysis, all the three NLE seem suspicious because they indicate that in the first two functions `ftpd_popen` is followed by `ftpd_pclose` along all paths highlighting the absence of error checking for `ftpd_popen`. And in function `retrieve`, NLE of zero indicates that `ftpd_popen` is not followed by `ftpd_close` along any path, which again is a violation. Upon checking, we found that the violation for `site_exec` function was a false-positive because trace generation had been terminated from exceeding the 10000 trace limit. And the violation for `retrieve` function was also a false-positive because `ftpd-pclose` was invoked through a function-pointer. But the violation in

function `statfilecmd` was a true-positive. The return value of `ftpd_popen` is indeed not checked for NULL. This is a new potential vulnerability found by FaultMiner in the latest version of wu-ftpd.

cups. In function `cupsPrintFiles2`, FaultMiner detected that while [$\_cupsglobals \rightarrow ippNew$] was present on 9 paths, [$\_cupsglobals \rightarrow ippNew \rightarrow ippAddString$] was present only on 8 paths. Also, there were six other functions that had a NLE of 1.0 for these sequences. On examining, we found that the six other functions were missing an error check on the return value of `ippNew` (which allocates a new printer request), which can return a NULL when `calloc` fails. Checking all the functions in `CUPS`, we found that while two calls to `ippNew` had NULL checks, there were seven other calls that did not check for NULL. These are serious violations because they can cause the program to crash under high loads and thus result in loss of unsaved state instead of a graceful degradation of service. This is a new potential vulnerability found by FaultMiner in the latest version of `CUPS`.

openssl. FaultMiner found a binary violation that while [$dtls1\_buffer\_message \rightarrow dtls1\_do\_write$] occurred in 12 functions, only `dtls1_do_write` was present in functions `dtls1_retransmit_message` and `dtls1_send_hello_request`. Upon inspection, we found that `dtls1_buffer_message`, which buffers a message for retransmission, is not necessary in the function `dtls1_retransmit_message` that does retransmission itself and is also not needed in the function `dtls1_send_hello_request` according to a comment we found in that function's body. This is an example of a semantic rule that is particular to a program's logic. This shows that FaultMiner can detect violations of such rules.

openssh. We did not find any interesting violations in the ones examined. But using FaultMiner, we were able to detect three previously known bugs in older versions of `openssh`. We reintroduced the faults in the latest version by commenting appropriate lines of code. We checked that the evidence used by FaultMiner to detect these faults were present in the older versions that had the faults as well. The first one was a memory leak bug where

Table 3.3
Violations detected for Property Two.

Software Evaluated	Events	Traces	Invariants Inferred	Violations Detected	Binary Violations	Time
wu-ftpd	765	375,609	171	0	3	4m57s
cups	1,696	215,804	244	0	4	7m53s
openssl	7,185	963,894	1,227	3	13	23m3s
openssh	4,956	440,923	704	12	5	7m26s

`xmalloc` was not followed by `xfree` along all the paths in the `toremote` function of `scp`. The second fault was a memory leak bug in `sshd` where `getrrsetbyname` was not followed by `freerrset` along all paths in the function `verify_host_key_dns`. The violation for this fault had a NLE of zero, which means that the function pair was used only in `verify_host_key_dns`. The third bug was a semantic bug detected by FaultMiner as a binary violation. A call to `packet_init_compression` was missing before the call to `buffer_compress_init_send` and `buffer_compress_init_recv` in function `packet_enable_delayed_compress` although there was evidence of this rule in two other functions `set_newkeys` and `packet_start_compression`. We had to use a *minsup* of 0.2 to detect this violation.

3.4.3 Property Two: Check-before-Use of Function Return Values

For this property, three types of events are considered as part of the EAM. *Call* events are generated at program points where there are calls to user-defined functions *with* a return value assignment. *ChkRetVal* events are generated at program points where return values corresponding to Call events are checked. *UseRetVal* events are generated where the return values are used. FaultMining is performed with two constraints for this property. The first constraint is that the events must correspond to the same function. The second constraint

is an ordering constraint on the type of events. A Call event should be followed by a ChkRetVal event and then by a UseRetVal event. Recall that these constraints are applied in the join phase of the apriori-generate algorithm.

In the current prototype implementation, we use lexical matching to correlate the Call, ChkRetVal, and UseRetVal events because we focus on lightweight analysis. ChkRetVal events are generated when the return value is part of a predicate in a conditional statement. UseRetVal events are generated when the return value is used in expressions or as arguments to function calls. Static analysis techniques such as pointer-analysis and def-use analysis can be used to further improve the accuracy of event information. The distinction between the static analysis phase that enables event generation and the rest of the phases in FaultMining is an important feature of our framework compared to related work. Stronger static analysis techniques can be applied to improve the results without having to modify the other phases.

Table 3.3[†] shows the number of events, traces, inferred invariants, detected violations, and time taken when FaultMiner was run with a *minsup* of 0.0, a *highconf* of 0.6, and *binary-support* of 2. Unlike property one for which we used a *minsup* of 0.4, for this property, we consider an event occurring even on a single path as evidence. This is because, the three types of events considered for this property generate so many different events that on applying the two constraints at *minsup* of 0.4, very few invariants are generated. Figure 3.14 shows that trace generation time overwhelmingly dominates the other phases for this property. We manually examined all the violations where the CLIs had all the three types of events but only the ChkRetVal event was missing in their CLI′s. We also examined binary violations for such CLIs. Table 3.3 shows statistics only for such violations. We discuss some of the interesting violations for this property detected by FaultMiner.

wu-ftpd. FaultMiner detected a missing check for the return value of `ftpd_popen` in `statfilecmd`. There was evidence for this binary violation in three other functions:

[†]Numbers reported for `openssh` are for the `sshd` component

`CheckSum`, `site_exec`, and `retrieve`. This is the same potential vulnerability that we found also as a violation of property one.

openssh. FaultMiner detected a previously known bug upon reintroducing it in the `sshd` code. The return value of a call to `session_new` was missing a NULL check before being used in function `do_authenticated1`. At the time the bug existed, there were two indications of evidence for this check in functions `session_open` and `mm_answer_pty`. This evidence enabled FaultMiner to detect this bug as a binary violation. This bug had existed in the code for more than four years before being corrected.

3.5 Obligations

Satisfying program invariants is an *obligation* on the part of a programmer to enforce correct behavior by a program. We borrow this terminology from [78], which proposes an analysis for identifying outstanding obligations and a language feature for automatically discharging obligations for error-handling defects in Java.

Syntactic obligations in C such as statements should end in a semicolon and open braces should have matching closing braces are checked automatically by the parser because they are mandatory to the construction of a valid program. However, identifying and discharging semantic obligations such as freeing used resources and performing appropriate checks is the duty of a programmer because such obligations cannot be generalized across all valid programs to be considered as mandatory and automatically enforced. This is especially the case when obligations are not related to known invariants for library functions and system calls.

Because FaultMiner can infer unknown invariants, it can be used to provide feedback on pending obligations and also to minimize obligations. This section describes these two applications of FaultMiner.

3.5.1 Identifying Pending Obligations

Pending obligations are those that are yet to be discharged during software development. FaultMiner can be used as part of the software development environment to continuously infer unknown invariants and use them, within the framework of the code editor for example, to suggest obligations that are to be discharged. Similar techniques exist for syntactic obligations in the form of syntax highlighting and syntax completion. However, identifying semantic obligations during the software development phase would require analyzing incomplete programs instead of whole programs. There is research in this area of analyzing incomplete programs [79, 80].

For example, lets assume that FaultMiner has determined that $F_a \rightarrow F_b$ is a likely invariant. Now if a programmer adds a call to F_a, and if FaultMiner determines that there is no succeeding call to F_b along a path, it adds *call* F_b as an obligation that has to be discharged along that path.

3.5.2 Minimizing Obligations

A programmer who wants to create a `chroot` "jail" has to remember to call `chdir("/")` after the call to `chroot`. If not, the jail is ineffective and creates a vulnerability. The programmer can be spared of this obligation if the two calls are combined into a single new API.

FaultMiner can be used to minimize the number of obligations in the context of unknown invariants. This can be done by analyzing the invariants inferred by FaultMiner and identifying invariants that can be coupled together. For example, if FaultMiner infers $F_a \rightarrow F_b$ as a likely invariant, and if it is possible to refactor the code so that $F_a \rightarrow F_b$ can be encapsulated in a single call to F_{ab} then it reduces the burden of obligation on the programmer to follow every call to F_a by a call to F_b. Instead the programmer can simply make a call to F_{ab}.

We considered sequences for the first property that had a confidence of one with respect to a subsequence in more than five functions. These represent likely invariants that had no violations (with respect to the subsequence) in more than five functions. We analyzed the source code of the programs that contained these invariants to determine if they could be refactored with little effort to minimize obligations by encapsulating the events in the invariant sequence in a meta-event i.e. a function that makes calls to the functions in the invariant. For example, we determined that the likely invariant [`cupsFileOpen` → `cupsFileGets` → `cupsFileClose`] in `cups` cannot be refactored trivially because the program logic in-between calls to these functions was different in different contexts. This was the case with the invariants [`packet_start` → `packet_put_cstring` → `packet_send` → `packet_write_wait`] and [`buffer_init` → `buffer_append` → `buffer_free`] of `openssh`.

In `openssh`, we found invariants of the form [`packet_get_string` → `packet_remaining` → `logit` → `packet_disconnect`]. Upon inspecting the source code, we found that [`packet_remaining` → `logit` → `packet_disconnect`] is part of a macro named `packet_check_eom` that should be called at the end of every message to check for remaining bytes. If there are bytes remaining after what `openssh` deems as the end of the message then it is considered as an integrity error and the connection is dropped. To our understanding, this is an important check and it is an obligation on the programmer to call this macro after reading the last string, character, or integer from the packet. We found 77 calls to this macro in the source code. We could instead make this check part of new APIs that are to be called by the programmer when reading the final bytes from the packet.

For `openssl`, we found the invariant [`dtls1_set_message_header` → `dtls1_buffer_message` → `dtls1_do_write`] which can be encapsulated into a new API that combines the creation, buffering, and writing of the message buffer. We note that the tradeoff between the effort to refactor the code and the benefits of minimizing the specific obligations for future programmer effort has to be considered. There has been extensive research in the area of software refactoring [81].

3.6 Challenges

FaultMiner is useful for detecting classes of defects for which the invariants are unknown. The invariants are inferred based on the premise that common behavior is correct behavior. Any deviation from common behavior is considered a violation. While we have shown that this is a useful technique, there are some challenges related to accuracy and efficiency that need to be overcome.

In static analysis based approaches to finding defects, approximations are used to make the analysis decidable, tractable, and practical. Such approximations result in false positives. For static approaches to finding defects related to unknown invariants, invariant generation is another source of false positives. Likely invariants might only be coincidences. Violations generated for such coincidental invariants are actually false positives. So false positives in our approach can be reduced by minimizing the number of coincidental invariants and by using more accurate static analysis techniques.

The challenge in minimizing coincidental invariants is in quantifying common and deviant behaviors (characterized by support and confidence). For example, we were able to detect the missing `packet_init_compression` in `openssh` only at a low support value of 0.2. The thresholds selected for these two attributes determine the number of violations reported. We do not believe that there is an ideal value for these attributes that will minimize the number of violations for any given program without missing the interesting ones. So instead of filtering violations based on these attributes, ranking violations in the decreasing order of support and confidence would be more useful.

A pathological case is when common behavior is incorrect behavior. This may happen when an invariant violation is introduced at several places in the program because of ignorance or because of replicating a single violation at multiple places as a result of copy-and-paste. While manual auditing can help in the first case, there are related data-mining approaches to detect copy-and-paste bugs [82].

Accurate and efficient techniques for flow-, context-, and path-sensitive pointer-analysis and dataflow analysis will enable more accurate event and trace generation in our ap-

proach. This will not only reduce the number of false positives but also enable us to check richer properties. For example, a well-known invariant is that, given the specifications of an untrusted source and a trusted sink, there should never be a tainting definition of a variable before its use at the sink without validating [*tainting-variable-definition→validate-variable→variable-use*]. Examples of this property include the dereferencing of user pointers in the kernel and format-string bugs. User pointers should be validated before dereferencing them in the kernel and user-supplied data should be validated before being used as format-string arguments. Failure to do so might result in a security violation. Unlike the dereferencing of user pointers in the kernel code or the use of user-supplied values as format strings, sinks, where tainted values must not be used without validating, might not always be known a priori. For example, there might be user-defined functions that perform sensitive operations and hence need to validate data that influence their operations. Such sinks can be identified if there is evidence of validation. If validation is performed on most paths except a few, then that might be evidence of deviant behavior and therefore a likely defect.

Trace generation along interprocedural paths is another challenge. All approaches so far consider only local paths although this may generate both false positives and false negatives. We are investigating approaches to make trace generation along interprocedural paths feasible. One solution is to collapse those path segments in the EAM that do not have any events associated with them. State-space reduction techniques used in model-checking may also be useful. Such optimizations however have to be reflected in the support and confidence calculations because a path is a unit of evidence in our approach. We are also exploring ways to reduce the memory footprint to allow us to consider more traces and more events per trace.

3.7 Related Work

We discuss related research in the four broad areas of specification-based defect detection, specification-annotation for defect detection, specification-inference for defect detection,

and the application of data mining techniques in computer security. We compare our work mainly with the static approaches to specification-inference for defect detection.

Specification-based Defect Detection. Given a specification of an invariant or its inverse—the defect signature, detection can be performed dynamically by observing program behavior at runtime or statically by observing the program's source or binary. There is considerable research in the application of dynamic and static analysis to finding software defects. Dynamic techniques [60, 83–85] and the static techniques of traditional dataflow analysis [11, 15–18], type systems [14], model checking [12, 19, 20], and abstract interpretation [21] have been used to detect software defects such as buffer overflows, format-string bugs, race-conditions, and memory leaks.

Specification-annotation for Defect Detection. Programmer annotations can be used to explicitly describe certain aspects of the specification in the program using a special annotation language. These are usually in the form of pre- and post-conditions. Although such approaches [86–89] are beneficial as part of the software development process, they require considerable programmer effort and cannot be automatically applied to legacy code.

Specification-inference for Defect Detection. There is some prior research in inferring specifications for the purpose of finding software defects. There are dynamic approaches that propose to infer specifications (mainly for supporting program evolution) by observing the runtime behavior of a program [22–26]. These approaches can observe actual runtime vales but have the same drawback as conventional testing in that they have to make inferences based on only the program paths that are exercised. They also require program instrumentation. Static approaches can observe all possible paths automatically by analyzing program text. We briefly discuss existing static approaches to defect detection by specification-inference.

Engler et al. [27] were the first to propose a static approach to inferring specifications from code and use them to find several bugs in Linux and OpenBSD. They refer to the inferred specifications as MUST and MAY *beliefs*. MUST beliefs are known invariants such as NULL pointers should not be dereferenced. MAY beliefs are likely invariants. The inferred MAY beliefs include function call pairs that should always occur together and function calls whose return values should be checked before use. Search space for function call pairs is reduced by considering only those functions that are related by dataflow or that have no arguments. In our work, we propose a general technique for inferring MAY beliefs across an arbitrary number of events. We have also shown that this technique can be instantiated to specific cases using constraints as in the case of the second property.

Weimer and Necula [28] infer function pairs (F_a, F_b) in Java programs where F_b occurs at least once in the cleanup code within a `catch` or `finally` block. The goal is to detect paths where F_a is not followed by F_b. The assumption is that specifications are most likely to be violated along exceptional paths. The constraint that F_b should be present inside an exception handler is used to limit the search space of function pairs.

More recently, Li and Zhou [29] proposed a general technique to infer implicit programming rules using data mining. However, their approach completely ignores control flow and considers the entire function body as a single path and so only binary-violations can be detected using their approach. Security violations typically occur on exceptional control flow paths. Our approach captures these more interesting violations besides the binary-violations. The distinction between the application of static analysis for event generation and the actual mining with support for constraints on event traces generated from EAM provides a more flexible framework for extending our approach to other security properties compared to their approach.

Data Mining in Security. Livshits and Zimmermann [90] recently proposed an interesting approach where they applied data mining on software revision histories to identify method calls that are frequently added to the code simultaneously. The assumption is that such method calls represent a common usage pattern. They combine this with a dynamic

analysis where they analyze the frequency of occurrence of the mined patterns and use that to classify deviations in usage as violations.

Data mining techniques have been used in the field of intrusion detection to learn "normal behavior" from training data and then flag deviations from that behavior in actual data as intrusions [34]. Anomaly detection approaches are appealing over signature-based approaches because they do not need an a priori characterization or specification of "bad behavior." But they often have a higher false positive rate because of the challenges in capturing normality. These advantages and drawbacks are common to analogous approaches in software defect detection. Identifying defects without an a priori knowledge of the correct behavior is a challenging task. What makes it even more difficult compared to anomaly intrusion detection is that there is no separate training data to help learn the correct behavior. The static or dynamic event traces represent both the training data and the real data. Data mining has also been used for other security applications including detection of malicious code [91].

Inferring likely invariants is also broadly related to the problems of inferring program dependences [92] and causal relationships [93] that have applications in areas such as information-flow analysis [94] and forensics [95].

4 CONCLUSIONS, CONTRIBUTIONS, AND FUTURE WORK

This dissertation explored two techniques to improve software assurance using lightweight static analysis. This chapter describes the conclusions reached, summarizes the major contributions, and outlines directions for future work.

4.1 Conclusions

In the context of host-based misuse detection systems, we proposed an efficient and scalable solution to the problem of constructing conservative approximations of legal program behaviors. 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. Deterministic markers improve runtime performance and increase resistance to mimicry attacks. The IAM construction algorithm has been shown to scale to a 54K line program with a substantial space overhead. We then showed how to reduce this overhead with automaton compaction techniques.

The second technique we proposed is FaultMining, a novel technique to detect unknown software defects. Program events are abstracted in an Event Automaton Model. Static analysis techniques can be used to generate program events. Temporal invariants on an arbitrary number of program events are inferred. The technique of inferring likely invariants and finding unknown defects is derived from well-known data-mining algorithms. Mining with constraints on event attributes is supported. We described how two types of security-critical program invariants can be inferred using this technique. We evaluated FaultMiner for these types of invariants on four widely-used security-critical real-world programs

namely `wu-ftpd`, `cups`, `openssl`, and `openssh`. We found two new potential vulnerabilities, four previously known bugs, and several other violations using FaultMiner and thus demonstrated that FaultMining is a useful and promising approach to finding violations of unknown invariants.

4.2 Contributions

In the context of host-based misuse detection systems, we have made the following contributions:

- Inlined Automaton Model. The IAM is a flow- and context-sensitive model which is as accurate as a PDA, up to recursion. We describe the construction of inlined automata and relate our results to previous work on context-sensitive models.
- Implementation. We describe a prototype implementation of IAM. 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. We evaluate the prototype on a benchmark suite that includes four real-world programs. We have shown reasonable runtime performance in all cases. We demonstrate scalability by monitoring a 54K line program.
- Deterministic Markers. We describe the concept of deterministic markers that can be used to convert the IAM model into a DFA. Our current prototype uses markers to reduce the non-determinism in the IAM model. This improves the runtime performance and the precision of the IAM model. We describe three different types of markers, outline their placement algorithms, evaluate the effect of markers on performance overhead, formalize their impact on space overhead, and illustrate the resistance to mimicry attacks imparted by them.

- Automata Compaction Techniques. We present and evaluate 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.

- Dynamic Checking. We describe an approach to and highlight the advantages of performing dynamic checking of program properties at the library interface in the context of our MDS implementation.

We have made the following contributions in the context of finding unknown software defects:

- Event Automaton Model. EAM is a program abstraction where events derived from a program's ICFG are modeled using a NFA. Event traces are generated using the EAM.

- Mining Likely Invariants. The FaultMiner algorithm is the first generalized control-flow-sensitive algorithm capable of analyzing temporal ordering of an arbitrary number and type of events. This enables us to mine complete likely invariants instead of multiple partial invariants. Algorithms for invariant inference and the subsequent detection of unknown software defects are derived from well-known data mining algorithms.

- Violations. We describe two types of invariant violations (non-binary and binary violations) and propose a violation ranking algorithm based on non-local evidence.

- Evaluation. We empirically evaluate the FaultMining technique by inferring two types of security-critical program invariants. We detected two new potential vulnerabilities and four known defects on analyzing violations of these invariants for four security-critical real-world programs.

- Obligations. Obligations are responsibilities of programmers to satisfy program invariants. We describe how FaultMiner can be used to identify pending obligations and minimize obligations on likely invariants.

4.3 Future Work

Static analysis based MDSs have the advantage of not generating false positives because of using conservative algorithms that over-approximate normal behavior. However, this affects the accuracy of such systems by introducing impossible paths in the models. This allows an attacker to evade the MDS using mimicry attacks thus resulting in false negatives. It will be interesting to explore the base-rate fallacy [96] in the context of static analysis based MDSs. It will also be useful to characterize the practical difficulty of constructing such mimicry attacks and to derive meaningful metrics for comparing the accuracy of MDSs.

We believe that inlining-based models are sufficient to model control-flow and that path-sensitivity and dataflow analysis are the next frontiers in this area of research. While the state-explosive effect of inlining cannot be avoided, especially for larger software artifacts, future work can benefit from identifying and including in models only a subset of the system calls that are considered as "dangerous." This categorization might have to be policy-specific in the absence of a generalized notion of which system calls are useful to an attacker.

Most research in this area has considered MDSs operating on uniprocessor systems. Desktop systems have started using dual-core processors and from Moore's law [97], one can expect that multi-core processors will be affordable in the next 5-10 years. It will be worthwhile to consider what constraints change and how MDS research can leverage the power of additional processors with associated resources and the resulting parallelism. For example, memory overheads might not be such an important concern and state exploration can be parallelized. Research in this direction has been initiated by [98], which proposes and evaluates a co-processor based misuse detection technique.

Research in the area of detecting unknown defects has to address concerns of accuracy, usability, and scalability. This will enable us to check for complex properties involving both control and dataflow.

LIST OF REFERENCES

LIST OF REFERENCES

[1] The Economic Impacts of Inadequate Infrastructure for Software Testing, May 2002. NIST Planning Report 02-3. Available from http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[2] IEEE.1990. ANSI/IEEE Standard Glossary of Software Engineering Terminology. IEEE Press.

[3] Bev Littlewood, Sarah Brocklehurst, Norman Fenton, Peter Mellor, Stella Page, David Wright, John Dobson, John McDermid, and Dieter Gollmann. Towards operational measures of computer security. *Computer Security*, 2:211–229, 1993.

[4] Ivan Victor Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, May 1998.

[5] Henry H. Feng, Jonathon T. Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[6] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[7] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th Annual Network and Distributed Systems Security Symposium*, February 2004.

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

[9] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[10] Rajeev Gopalakrishna, Eugene H. Spafford, and Jan Vitek. Efficient intrusion detection using automaton inlining. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 18–31, 2005.

[11] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.

[12] Hao Chen and David Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the Ninth ACM Conference on Computer and Communications Security*, 2002.

[13] Alan DeKok. PScan: A limited problem scanner for C source files. Available at http://www.striker.ottawa.on.ca/∼aland/pscan.

[14] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[15] Ken Ashcraft and Dawson R. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[16] Dawson R. Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Symposium on Operating Systems Principles*, 2003.

[17] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[18] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2000.

[19] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[20] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, 2004.

[21] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[22] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[23] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2), 2001.

[24] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[25] Jinlin Yang and David Evans. Automatically inferring temporal properties for program evolution. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004.

[26] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 15–26, June 2005.

[27] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[28] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[29] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[30] Henry H. Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2003.

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

[32] Anita K. Jones and Yu Lin. Application intrusion detection using language library calls. In *Proceedings of the 17th Annual Computer Security Applications Conference*, 2001.

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

[34] Wenke Lee, Salvatore J. Stolfo, and Kui W. Mok. A data mining framework for building intrusion detection models. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1999.

[35] R. Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[36] Andreas Wespi, Marc Dacier, and Hervé Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, 2000.

[37] Calvin Ko, George Fink, and Karl N. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, 1994.

[38] Kymie M.C. Tan, Kevin S. Killourhy, and Roy A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Proceedings of the Recent Advances in Intrusion Detection*, 2002.

[39] Kymie M.C. Tan, John McHugh, and Kevin S. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Proceedings of the Fifth International Workshop on Information Hiding*, 2002.

[40] David Wagner and Paolo Soto. Mimicry attacks on host based intrusion detection systems. In *Proceedings of the Ninth ACM Conference on Computer and Communications Security*, 2002.

[41] Lap-Chung Lam and Tzi cker Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Proceedings of the Recent Advances in Intrusion Detection*, pages 1–20, 2004.

[42] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, Second edition*. ACM Press, 2001.

[43] Benjamin A. Kuperman and Eugene H. Spafford. Generation of application level audit data via library interposition. CERIAS TR 99-11, COAST Laboratory, Purdue University, October 1998.

[44] PROLANGS Analysis Framework. http://www.prolangs.rutgers.edu/public.html.

[45] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of 11th International Conference on Compiler Construction*, 2002.

[46] GCC 3.3.3 Manual. http://gcc.gnu.org/onlinedocs/gcc-3.3.3/gcc/.

[47] Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. In *Proceedings of the Summer USENIX Conference*, pages 267–278, 1994.

[48] Benjamin A. Kuperman. *A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources*. PhD thesis, Purdue University, 2004.

[49] Dale Dougherty and Arnold Robbins. *sed & awk*. O'Reilly Media, Inc., second edition, 1997.

[50] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[51] Haizhi Xu, Wenliang Du, and Steve J. Chapin. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In *Proceedings of the Recent Advances in Intrusion Detection*, pages 21–38, 2004.

[52] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. On effective model-based intrusion detection. TR 1543, University of Wisconsin, Madison, 2005.

[53] The PaX project. pax.grsecurity.net/.

[54] Gaurav S. Kc and Angelos D. Keromytis. e-NeXSh: Achieving an effectively non-executable stack and heap via system-call policing. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 286–302, Washington, DC, USA, 2005. IEEE Computer Society.

[55] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 272–280, 2003.

[56] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 281–289, 2003.

[57] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, New York, NY, USA, 2002. ACM Press.

[58] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[59] Timothy K. Tsai and Navjot Singh. Libsafe 2.0: Detection of format string vulnerability exploits, February 2001. White Paper Version 3-21-01, Avaya Labs, Avaya Inc.

[60] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Conference*, 2001.

[61] Michael F. Ringenburg and Dan Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 354–363, New York, NY, USA, 2005. ACM Press.

[62] Kumar Avijit, Prateek Gupta, and Deepak Gupta. TIED, LibsafePlus: Tools for runtime buffer overflow protection. In *Proceedings of the USENIX Security Symposium*, pages 45–56, 2004.

[63] Timothy K. Tsai and Navjot Singh. Libsafe: Transparent system-wide protection against buffer overflow attacks. In *Proceedings of the International Conference on Dependable Systems and Networks*, page 541, 2002.

[64] Neils Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[65] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the USENIX Security Symposium*, pages 171–190, 2002.

[66] Jonathon T. Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P. Miller. Environment-sensitive intrusion detection. In *Proceedings of the Recent Advances in Intrusion Detection*, pages 185–206, 2005.

[67] Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. Dataflow anomaly detection. To appear in IEEE Symposium on Security and Privacy, Oakland, CA, May 2006.

[68] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-Oriented Programming Languages, Systems and Applications*, 2000.

[69] Haizhi Xu. *Surviving Malicious Code Attacks*. PhD thesis, Syracuse University, 2006.

[70] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[71] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 2000.

[72] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, 1993. ACM Press.

[73] Lee Garber. New chips stop buffer overflow attacks. *IEEE Computer*, 37(10):28, 2004.

[74] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, 1995.

[75] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. Technical Report RJ9910, IBM Almaden Research Center, 1994.

[76] CVE-2004-0077. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2004-0077.

[77] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.

[78] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the Conference on Object-Oriented Programming Languages, Systems and Applications*, pages 419–431, 2004.

[79] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, LNCS 1687, pages 235–252, 1999.

[80] Atanas Rountev. *Dataflow Analysis of Software Fragments*. PhD thesis, Rutgers University, August 2002.

[81] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[82] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, 2004.

[83] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Seventh USENIX Security Conference*, 1998.

[84] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *Proceedings of the 10th USENIX Security Conference*, 2001.

[85] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access error. In *Proceedings of the USENIX Winter Technical Conference*, 1992.

[86] Brian Chess. Improving computer security using extended static checking. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[87] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 1998.

[88] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1994.

[89] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Conference*, 2001.

[90] Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[91] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[92] Mark Weiser. Program slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 439–449, 1981.

[93] Judea Pearl. *Causality: models, reasoning, and inference*. Cambridge University Press, 2000.

[94] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[95] Brian D. Carrier. *A hypothesis-based approach to digital forensic investigations*. PhD thesis, Purdue University, 2006.

[96] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security*, 3(3):186–205, 2000.

[97] Moore's Law. http://www.intel.com/museum/archives/history_docs/mooreslaw.htm.

[98] Paul D. Williams. *CUPIDS: Increasing information system security through the use of dedicated co-processing*. PhD thesis, Purdue University, 2005.

VITA

VITA

Rajeev Gopalakrishna obtained the B.E. degree in Computer Science and Engineering from the Regional Engineering College (now known as National Institute of Technology), Tiruchirappalli, India in 1999. He received the M.S. degree in Computer Sciences and the Ph.D. degree in Computer Sciences from Purdue University in 2001 and 2006 respectively. Rajeev's research interests are primarily in information assurance.