CERIAS Tech Report 2006-07

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

by Rajeev Gopalakrishna, Eugene H. Spafford, and Jan Vitek

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

FaultMiner: Discovering Unknown Software Defects using Static Analysis and Data Mining

Abstract

Improving software assurance is of paramount importance given the impact of software on our lives. Static and dynamic approaches have been proposed over the years to detect security vulnerabilities. These approaches assume that the signature of a defect, for instance the use of a vulnerable library function, is known apriori. A greater challenge is detecting defects with signatures that are not known apriori – unknown software defects. In this paper, 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. We found two new potential vulnerabilities, four previously known bugs, and several other violations. This suggests that FaultMining is a useful and promising approach to finding unknown software defects.

1 Introduction

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 → denotes a happens-before relationship): [`isnull(ptr)` → `*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` → `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` → `chdir("/")`].

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 paper 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

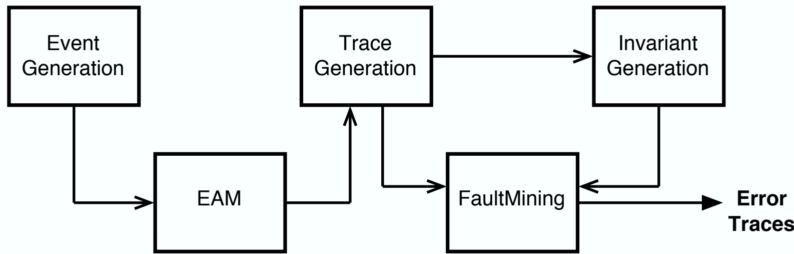

Figure 1: FaultMiner Framework.

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 think of 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 apriori. 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 [5, 18, 21, 33], which observe a program's runtime behavior, and *static* approaches [16, 26, 32], 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. [16] and Weimer and Necula [32] consider only function pairs in their invariants. Li and Zhou [26] ignore control flow from conditional statements and consider the function body as a single path. FaultMiner overcomes these limitations.

In this paper, we propose a general approach to finding unknown defects. An overview of the FaultMiner framework is illustrated in Figure 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 paper is organized as follows. Section 2 describes event generation, EAM, and trace generation. Section 3 explains invariant generation. Section 4 describes the FaultMining technique and the two security properties. Extensive experimental evaluation is presented in Section 5. Sections 6 and 7 discuss the challenges that need to be overcome in future and related work respectively and Section 8 presents our conclusions.

2 Event Automaton Model

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.

For our purposes, an EAM is a Non-deterministic Finite Automaton 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 fi lter 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 2. Trace generation for this EAM produces two traces [`foo`] and [`bar` → `foo`]. Practical considerations with trace generation in FaultMiner are discussed in Section 5.1.

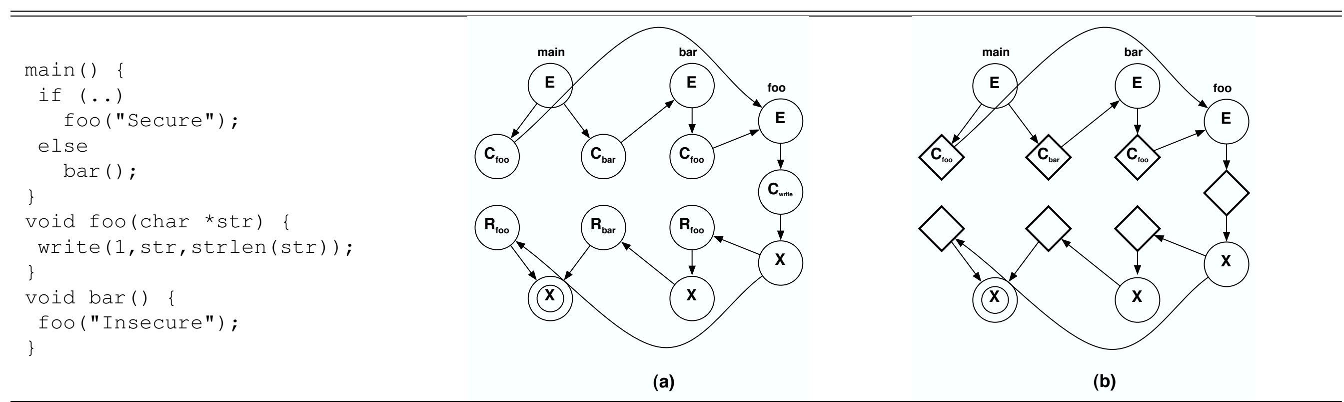

Figure 2: An example program. (a) ICFG representation of the program. E, X, C, and R represent the entry, exit, call, and return nodes respectively. (b) EAM representation for user-defi ned function invocations.

3 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 [4]. The terminology used in the FaultMiner algorithm is defined below.

Definition 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 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 *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 4 *An event sequence with support greater than user-defined minimum support value, minsup, is called a large event sequence.*

Definition 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* s.t. $0 <$ *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 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 [3] provide a proof of correctness of this candidate generation algorithm. The AprioriAll and apriori-generate algorithms are presented in Figure 3.

```
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} ≠ ∅ 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
```

```
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: The FaultMiner AprioriAll Algorithm. L_k represents the set of of all large k-sequences. C_k represents the set of candidate k-sequences.

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 effi ciency.

4 FaultMining

FaultMining is based on the two concepts of suffi cient evidence of common behavior and suffi cient evidence of deviant behavior. A program path or an event trace is the unit of evidence in FaultMiner. Suffi cient 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 just enough to classify it as deviant and not great enough 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 4 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.

Defi nition 6 *A complete likely invariant (CLI) is a sequence $\chi \in$ set of large sequences S generated by AprioriAll, s.t. $\nexists$ any other sequence $\rho \in S$ that satisfi es$\chi \sqsubset \rho$. A partial likely invariant (PLI) is a sequence $\phi \in S$ s.t. $\exists\ \chi$ ($\chi \neq \phi$) that satisfi es$\phi \sqsubset \chi$.*

This is a signifi cant improvement over related approaches [16, 32] because PLIs might not be meaningful when

```
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
```

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

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

Figure 4: FaultMiner Algorithm.

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 4. 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 [16, 32] consider the simplest case of temporal ordering of two events and use ad-hoc techniques to limit the search space. Li and Zhou [26] 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.

4.1 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. Such unknown defects resulting from a violation of 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 [16, 32] 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 [26] 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 [1]. 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.

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

Table 1: Characteristics of Evaluated Software.

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.

5 Evaluation

We have implemented our FaultMiner tool using CIL [28]. 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 1. The experiments were run on a 2.8 GHz Linux machine with 1 GB RAM.

5.1 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 [26] 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

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

Table 2: Violations Detected for Property 1.

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 algorithm. This is an expensive operation and needs to be performed effi ciently. For this reason, a*hash-tree* data-structure is used [2, 4].

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 specifi es the number of buckets in the interior nodes, and the*leaf-threshold*, that specifi es 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 [2, 4] for a description of the algorithms for hash-tree insertion and support calculation.

5.2 Property 1: Function Call Sequences

For this property, calls to user-defi ned 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

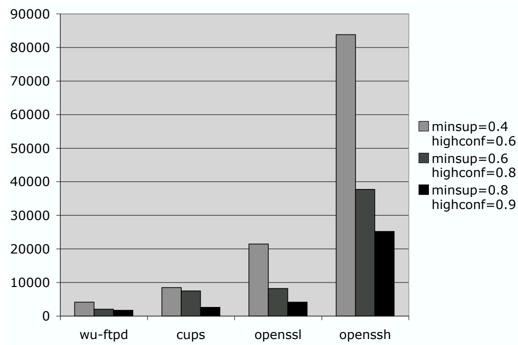

Figure 5: Number of Invariants Generated for Property One at Three Levels of Support and Confidence.

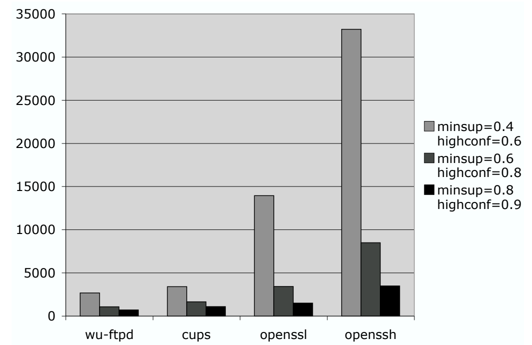

Figure 6: Number of Violations Generated for Property One at Three Levels of Support and Confidence.

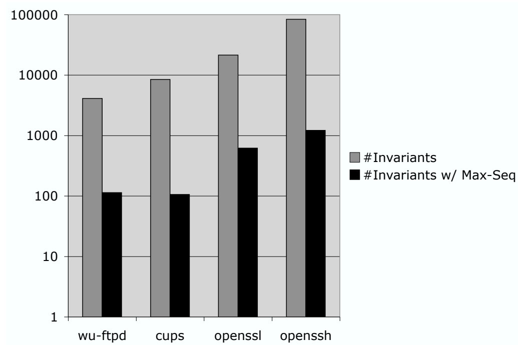

Figure 7: Number of Invariants Generated for Property One Without and With Running the Maximal-Sequences Algorithm.

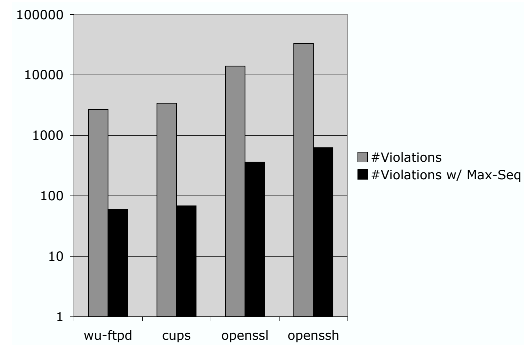

Figure 8: Number of Violations Generated for Property One Without and With Running the Maximal-Sequences Algorithm.

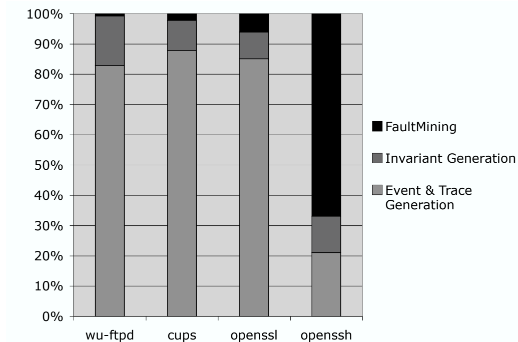

Figure 9: Percentage Distribution of Time Among the Different Stages of FaultMiner for Property One.

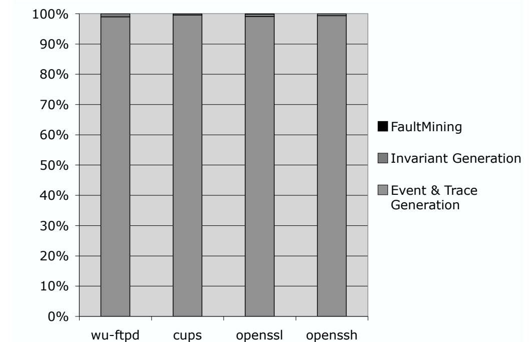

Figure 10: Percentage Distribution of Time Among the Different Stages of FaultMiner for Property Two.

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 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 5 and 6 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 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 7 and 8 (note that the y-axis is in logarithmic scale). Figure 9 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 [$_cupsglobals \rightarrow ippNew \rightarrow ippAddString$] was present on 9 paths, [$_cupsglobals \rightarrow ippNew$] 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

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

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

Table 3: Violations Detected for Property 2.

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 `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.

5.3 Property 2: 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. 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[†] 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

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

the two constraints at *minsup* of 0.4, very few invariants are generated. Figure 10 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 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: `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.

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 [25].

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 approach. 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 sanity-checking [*tainting-variable-definition→sanity-check-variable→variable-use*]. Examples of this property include the dereferencing of user pointers in the kernel and format-string bugs. User pointers should be sanity-checked before dereferencing them in the kernel and user-supplied data should be sanity-checked 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 sanity-checking, might not always be known apriori. For example, there might be user-defined functions that perform sensitive operations and hence need to sanity-check data that influence their operations. Such sinks can be identified if there is evidence of sanity checking. If sanity checking 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.

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. Space constraints prevent us from a more detailed comparison.

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 [12, 13, 14, 22] and the static techniques of traditional dataflow analysis [6, 7, 17, 20, 31], type systems [30], model checking [9, 10, 34], and abstract interpretation [8] 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 [11, 15, 19, 23] 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 [5, 18, 21, 33]. These approaches 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. [16] 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 this paper, 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 [32] 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 [26] 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 [27] 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 [24]. Anomaly detection approaches are appealing over signature-based approaches because they do not need an apriori 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 apriori 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 [29].

8 Conclusion

We have proposed FaultMining, a novel technique to detect unknown 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 have described how two types of security-critical program invariants can be inferred using this technique. We have evaluated FaultMiner for these types of invariants on four widely-used security-critical real-world programs namely `wu-ftpd`, `cups`, `openssl`, and `openssh`. We have found two new potential vulnerabilities, four previously known bugs, and several other violations using FaultMiner and thus have demonstrated that FaultMining is a useful and promising approach to finding violations of unknown invariants.

References

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

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

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

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

[5] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifi cations. In*Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[6] 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.

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

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[9] 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 (NDSS)*, 2004.

[10] Hao Chen and David Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, 2002.

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

[12] 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-overfbw attacks. In *Proceedings of the 7th USENIX Security Conference*, 1998.

[13] 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.

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

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

[16] 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 (SOSP)*, 2001.

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

[18] 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.

[19] 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 (FSE)*, 1994.

[20] 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 (CCS)*, 2003.

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

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

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

[24] 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.

[25] 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 6th Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[26] 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 (FSE)*, 2005.

[27] 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 (FSE)*, 2005.

[28] 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 (CC)*, 2002.

[29] 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.

[30] 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.

[31] 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 (NDSS)*, 2000.

[32] 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 (TACAS)*, 2005.

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

[34] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*, 2004.