MAGIC: Path-Guided Concolic Testing

Zhanqi Cui[†, ‡, §], Wei Le[§], Mary Lou Soffa[§], Linzhang Wang[†, ‡] and Xuandong Li[†, ‡]
[†]State Key Laboratory of Novel Software Technology, Nanjing University, P. R. China
[‡]Department of Computer Science and Technology, Nanjing University, P. R. China
[§]Department of Computer Science, University of Virginia, USA
zqcui@seg.nju.edu.cn; {weile, soffa}@cs.virginia.edu; {lzwang, lxd}@nju.edu.cn

Abstract

Concolic testing has been proposed as an effective technique to automatically test software. The goal of concolic testing is to generate test inputs to find faults by executing as many paths of a program as possible. However, due to the large state space, it is unrealistic to consider all of the program paths for test input generation. Rather than exploring the paths based on the structure of the program as current concolic testing does, in this paper we generate test inputs and execute the program along the paths that have identified potential faults. We present a path-guided testing technique that combines path-sensitive static analysis with concolic testing. The program under test is statically analyzed before testing to find potential faults (suspicious statements) and corresponding suspicious path segments. Then the program is tested, guided by static information, to avoid generating test inputs for safe paths. A tool, MAGIC, has been implemented based on our technique to test for buffer overflow. We have experimentally evaluated MAGIC on a set of C benchmarks, and the results show that compared to concolic testing, MAGIC found about 2.5 times more faults, and using the path information, MAGIC triggers the faults 25.3 times faster on average for a set of benchmarks.

1. Introduction

To improve the quality of software before release, dynamic testing and static analysis are the most widely used techniques for detecting faults [19]. Static analysis can find faults at the implementation stage, when fixing a fault is relatively cheap [8]. In addition, static analysis is cost effective, as it does not require the execution of a program. One major drawback of static analysis is that it potentially reports false positives, preventing real and important faults to be found. Dynamic testing, on the other hand, is effective in avoiding false positives and confirming realistic bugs with concrete test inputs [2]. Using the runtime information, we are able to debug the program to fix the fault. However, most dynamic testing techniques need an adequate test suite that can trigger faults, which is not available in most cases.

To address the problem of generating an adequate test suite, concolic testing [12] [16] has been proposed to automatically test software, with the goal of exploring as many paths of a program as possible. In concolic testing, the program under test is concretely executed and symbolically evaluated simultaneously. Instrumentation is inserted to the program to collect the symbolic path constraints and value updates during program execution. The symbolic constraints are solved to generate test inputs targeting a new path. When symbolic values cannot be collected, symbolic expressions are simplified by using the corresponding concrete values. Concolic testing terminates either when 1) no more new paths can be further executed due to incapability of solving complex constraints, 2) all of the paths in a program have been executed, or 3) a time threshold is reached. Considering that there is an exponential number of paths, often only a small portion of the program paths are actually covered by concolic testing [12] [16].

Our insight is that rather than executing as many paths as possible to trigger faults, we should only execute paths that have been identified as potentially containing faults. That is, we should guide the concolic testing to generate test inputs along paths which have potential faults. Recently, a tool, Marple [13] has been developed to identify both the statements where a fault potentially occurs (which we call *suspicious statements*), and the path segments that can produce the faults (which we call *suspicious path segments*). In Marple, program paths are categorized into four types: *infeasible*, *safe*, *vulnerable*, and *don't-know*. *Don't-know* paths are those that static analysis cannot determine due to the presence of library calls, complex pointers or loops. Both the vulnerable and don't-know paths are considered as *suspicious* paths. Analyzing a program using Marple before testing provides suspicious statements and correspond-

ing suspicious path segments. The advantages of combining static analysis and dynamic testing are twofold: on one hand, the suspicious information provided by static tools can be used to guide dynamic testing, and on the other hand, the runtime information provided by dynamic testing can be used to confirm and refine the results of static analysis.

In this paper, we present a hybrid technique to automatically find software faults, where static analysis is first applied to identify suspicious statements and path segments, and test inputs are then generated for these suspicious paths to trigger the faults. A novelty of our work is that our technique is path based, i.e., we direct dynamic testing to the path segments rather than a program point. Given only a suspicious program point, both safe and infeasible paths can traverse it [13], and efforts for generating test inputs for these paths are not useful for triggering faults. Compared to program points, path information is more precise, and can help further reduce the search space for test input generation.

This research addresses three challenges. Considering that the number of suspicious paths can still be huge, we need to develop a representation of path information used in testing. Also, static analysis produces false positives and negatives. We need to understand the impacts of the potential imprecision in guiding test input generation. Furthermore, not every execution that exercises a faulty path necessarily triggers the fault; besides path constraints, we also need to track fault conditions for test input generation.

Our technique proceeds in three steps. First, the program under test is analyzed by a path-sensitive static analysis tool. Both the suspicious statement and corresponding path segments along which a fault could occur are identified, represented using a *path graph*. Second, reachability relationships from each branch to these path segments are computed. In the third step, we execute the program with an initial input, and use the reachability information and the path graph to select the paths of interest. During execution, we generate test inputs that 1) can reach a suspicious statement along a corresponding suspicious path segment, and 2) can trigger the fault condition at the suspicious statement.

We have implemented our techniques in a tool called MAGIC (MArple-GuIded Concolic testing). Currently, this tool handles buffer overflow for C program; however the technique is applicable for multiple types of faults, including both data-centric, e.g., integer overflows, and control-centric faults such as memory leaks [14]. The static information is provided by our path-sensitive static analyzer, Marple, and the dynamic testing components are built based on concolic testing technique. We have experimentally evaluated MAGIC on four diverse C benchmarks. The experimental results show that compared to concolic testing, MAGIC can detect more bugs with less runtime overhead.

The main contributions of this paper include:

- automatic test input generation to exploit statically identified faults,
- application of static path information for reducing the cost of dynamic testing,
- the implementation of the techniques for detecting buffer overflows, and
- an experimental study that demonstrates the effectiveness of our technique.

In Section 2, we use an example to intuitively explain the technique. In Section 3, we present an overview of MAGIC. In Sections 4 and 5, we give detail designs of static and dynamic components. Section 6 shows our experimental results, and Section 7 compares the related work. We give conclusions in Section 8.

2 An Example

In Figure 1, we show an example adapted from the benchmark WuFTP-1 [20]. This example contains three paths and two buffer write statements at lines *6* and *10* respectively. A buffer overflow exists at line *10*. Using this example, we show how concolic testing and our technique find this buffer overflow.

Applying concolic testing for buffer overflow [18], we first execute the program with an initial input. Without loss of generality, we assume in the first run, *argc=1*, which means that no command line argument is supplied to the program. Under this input, the program takes the execution path $\langle 2, 3, 4(T), 5\rangle$. During execution, the symbolic path constraint [*argc !=2*] is collected. As the goal of concolic testing is to cover as many paths as possible, in the second run, the tester inverts the path constraint to [*argc=2*], aiming to exercise the branch *4(F)*. Suppose a command line argument *"a"* is generated for *argv[1]*. Running this input, path $\langle 2, 3, 4(F), 6, 7, 8(F), 10\rangle$ is taken. Along this path, the tester checks the buffer safety at lines *6* and *10*, and determines that both lines 6 and 10 are safe for this execution. Meanwhile, the tester also derives that line 10 can be an overflow if the length of argv[1] is larger than 8. Using this buffer overflow condition, the tester can generate an input "aaaaaaaaa" for *argv[1]*, which leads the execution to path $\langle 2, 3, 4(F), 6, 7, 8(F), 10\rangle$, and exploits the buffer at line 10. Since there are still paths that have not been covered, the concolic testing continues to invert the path constraint at line 8, aiming to take branch *8(F)*. A string *"."* is generated as the input for *argv[1]* to exercise $\langle 2, 3, 4(F), 6, 7, 8(T), 9\rangle$. For this example, concolic testing generates a total of three test inputs and covers all the three paths of the program. Buffer write statements at lines 6 and 10 are checked for each path that exercises them.

Our observation is that not all of the buffer write statements are equally suspicious for buffer overflows. Even for

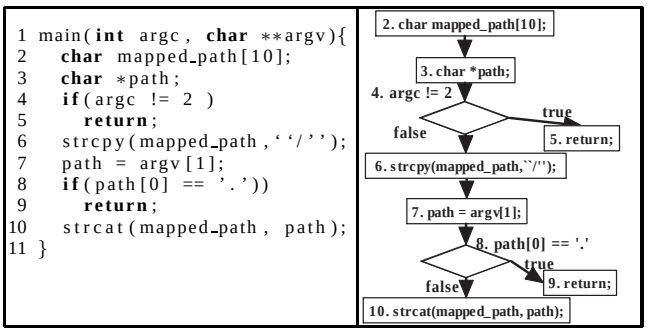

Figure 1. An Example

a suspicious statement, not all the paths that traverse it are faulty. To save the cost of test input generation, we should direct the testing along suspicious paths.

Applying our technique, we first statically identify that line *10* is suspicious for buffer overflow along path segment $\langle 6, 7, 8(F), 10\rangle$, and line *6* is safe, which implies that no checks are needed for this statement at run time. We then perform a reachability analysis, and find that branch *4(F)* reaches the suspicious path segment, but branch *4(T)* cannot. Based on the above static information, we run a concolic testing. The program is first executed with no arguments along path $\langle 2, 3, 4(T), 5\rangle$. As branch *4(F)* can reach the suspicious path segment, we inverse the symbolic path constraint and generate an input *"a"*. Under this input, the program executes $\langle 2, 3, 4(F), 6, 7, 8(F), 10\rangle$. Since the suspicious path segment is traversed, the tester determines if the buffer overflow is triggered. As the buffer overflow is not triggered under this input, the tester integrates the buffer overflow condition at line *10*, and generates a new input *"aaaaaaaaaa"* to exploit the buffer overflow.

With the static information, we do not need to explore the program nodes that cannot reach the suspicious path segment, e.g., branch *4(T)*. Only paths that cross a suspicious path segment are checked for buffer overflow. For instance, no effort is needed to generate test input for path $\langle 2, 3, 4(F), 6, 7, 8(T)\rangle$. Testing can be terminated early when the potential faults are triggered. We exploit the overflow at line *10* by only generating two test inputs, and the possibility of buffer overflow is checked only once along one path.

3 An Overview

This section provides a high level description of MAGIC, including the components of MAGIC and their interactions.

3.1 Components

MAGIC consists of five components, shown in Figure 2. Marple and the reachability analyzer are the two static components. Marple is a static path-sensitive analyzer that reports the suspicious statements as well as the suspicious path segments. The reachability analyzer calculates reachability relationships from each branch of the program to the suspicious path segments. The dynamic testing components are built based on concolic testing, including a program instrumentor, a test input generator and a test driver. The program instrumentor inserts statements to the program to collect symbolic constraints and concrete values during testing. The test input generator generates test inputs using symbolic path constraints and fault conditions. The test driver executes the program with test inputs and performs symbolic evaluation simultaneously.

Our testing components make several improvements on traditional concolic testing. First, we use boundary values to initiate the test input, which is experimentally shown to achieve a better branch coverage than using a fixed given value as the input. Another enhancement is that we dynamically change the program state at runtime when a fault is perceived to avoid the crash of the program; otherwise, manual effort has to be involved to fix the fault before testing can continue. Furthermore, we model program operations that are potentially related to the production of a certain type of fault. For instance, to trigger a buffer overflow, we handle string libraries and pointer operations. Concolic testing might never be able to exercise desired paths if these operations are not modeled. In addition to path constraints, we also construct fault conditions for test input generation. The goal is to ensure the generated inputs not only can exercise a desired path but also trigger faults.

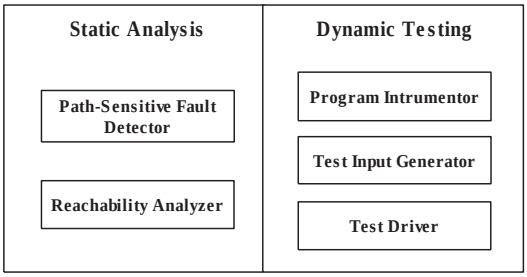

Figure 2. Components of MAGIC

3.2 Workflow of MAGIC

As shown in Figure 3, MAGIC first statically analyzes the program and reports suspicious statements and path segments. Based on the program source and the path segments, MAGIC runs a reachability analysis to determine, for each branch, whether the execution at the branch is able to reach any of the suspicious path segments. MAGIC instruments the program to collect information needed at runtime. Testing runs on the instrumented program with an initial test

input. During execution, the tester determines whether the current execution can traverse any suspicious path segment. Meanwhile, the tester collects concrete and symbolic values; when a suspicious fault is encountered, the symbolic constraints regarding path constraints and fault conditions are solved by a constraint solver for potential test inputs. Testing terminates when a program input is discovered that can trigger the fault, or the paths that traverse the set of suspicious path segments are all examined, which show that the suspicious statement is likely safe along the reported suspicious path segments. The details of static and dynamic components are presented in Sections 4 and 5.

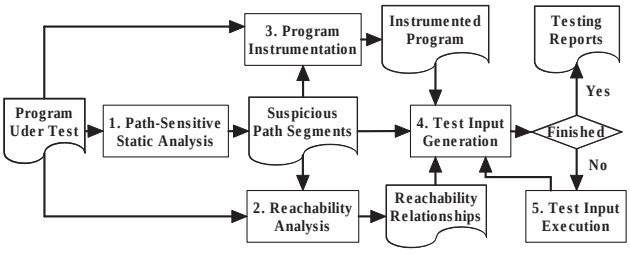

Figure 3. The workflow of MAGIC

4. Path-Based Static Analysis

We first describe the static components of MAGIC, including Marple and the reachability analyzer. The focus is to explain how the static analysis works and also the choice of static information provided to dynamic testing.

4.1 How Marple works

To guide dynamic testing, we need both the suspicious statements and the path information about the statements. We require that 1) the static information should be as precise as possible, as high false negatives in static results can lead to missed faults, while high false positives can result in extra overhead by directing dynamic testing along safe paths; and 2) the static analyzer should achieve reasonable coverage, which means the fault detection should be performed along as many paths as possible. Marple is used not only because it provides the desired information, but also it has reasonable precision and coverage [13].

Exhaustively identifying path information incurs high overhead. Marple applies a demand-driven algorithm to address the scalability. In Figure 4, we show how a demand-driven algorithm works to identify suspicious path segments. First, Marple scans the program source and constructs queries at the statements where a fault could happen. For example, to detect buffer overflow, we raise a query at the library calls *strcpy* and *strcat*, or a direct buffer

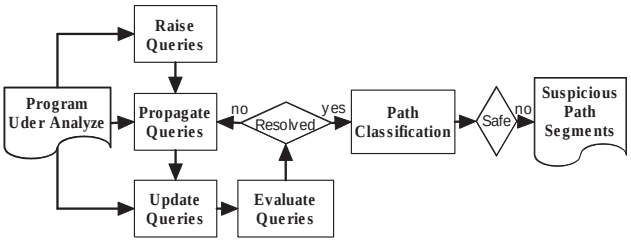

Figure 4. Marple

assignment *a[i]='x'*. The query asks whether the safety constraints at the statement can be satisfied. In the case of buffer overflow, the query is whether the string length at the current program point can be larger than the buffer size. The query is propagated backwards towards the beginning of the program along the paths reachable from the suspicious statement. During propagation, the relevant values or ranges of program variables are collected to resolve the query. The propagation of a query terminates when an infeasible path segment is encountered (we run an infeasible path detection before fault detection). The propagation of a query also terminates when a resolution is reached under one of the two cases: 1) the collected information is sufficient to determine if the safety constraints can be satisfied, or 2) don't-know factors are encountered such as library calls, complex loops or pointer arithmetics. The statement where the propagation of a query terminates is called a *resolution point*.

Based on the resolution of a query, the path segments traversed by the query are categorize into *infeasible*, *safe*, *vulnerable*, and *don't-know*. Vulnerable and don't-know paths will be supplied to guide the dynamic testing. Further details of Marple are provided in the paper [13], including the techniques that handle the branches, loops and procedures, as well as optimizations to further speedup the analysis.

4.2 Representing Path Information

To determine what path information we should provide to dynamic components, we first need to understand the semantics of two types of suspicious path segments. In Marple, a path segment is determined as vulnerable if: 1) along the path segment, the fault always occurs independent of program inputs, e.g., a buffer overflow with a constant string, or 2) there exists an entry point along the path segment, where users can supply an input to trigger the fault, e.g., a buffer overflow with an external string. As its determination is independent on any other information beyond the path segment, any execution that traverses the vulnerable path segment (with a proper input supplied at the entry point along the path segment for the second case) can trigger the fault. A don't-know path segment is determined when

the query encounters don't-know factors. If the don't-know factors are resolved, the query is potentially propagated further before being determined as vulnerable, in which case, the don't-know path segment can be viewed as a partial vulnerable path segment. Some of the don't-know paths can be safe and thus executions along don't-know paths do not necessarily trigger the fault.

There are choices on the number of suspicious path segments we should present. In testing, we only need to demonstrate the exploits of the buffer overflow along one execution. However, presenting one path segment for test input generation is not sufficient. The reasons are twofolds. First, static information can be imprecise. For example, even a buffer overflow potentially occurs at the statement, a suspicious path segment randomly picked from the static results can be infeasible. Although we have applied a static analysis to remove some of the infeasible paths, infeasible path detection is an undecidable problem, and we can not remove all of them for a program. The second reason is that concolic testing is not always able to generate a test input to exercise a given path, as some of the symbolic constraints are too complex to solve. We also can choose to enumerate all of the suspicious path segments; however, this solution is not scalable as there potentially exists a large number of suspicious path segments, and both storing and accessing them at runtime can incur unacceptable overhead. There is also the choice of using a fixed number of path segments. The challenge is to determine a reasonable number and also strategies to select the path segments.

In this paper, we developed a *path graph* to represent a set of suspicious path segments that end at the same suspicious statement. A path graph is a directed graph *(N, E)*, where:

- *N* is a finite set of nodes. Each node is a program statement that occurs in a suspicious path segment. The graph has multiple entries, and each resolution point is an entry. The graph has only one exit, which is the suspicious statement.
- *E* is a finite set of the directed edges. For any pair of statements $\langle l_i, l_j \rangle$ found along a suspicious path segment, where l_i is the predecessor of l_j, we add an edge to the graph from l_i to l_j.

As an example, we show in Figure 5 (a) and (b), two suspicious path segments ending at the same suspicious statement *r*. s_1 and s_2 are two resolution points. Figure 5(c) is the path graph constructed for the suspicious path segments in (a) and (b).

The path graphs are generated by Marple. Each path graph corresponds to a fault. In Marple, computation of path graphs is a forward analysis following the fault detection. As described above, in fault detection, queries are propagated backwards for resolutions. During propagation,

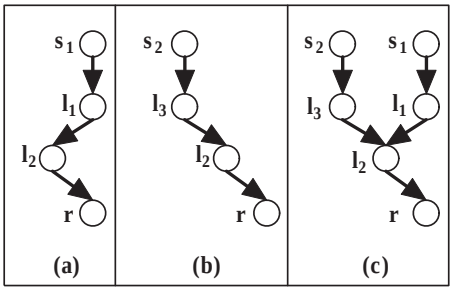

Figure 5. A path graph for two suspicious path segments

queries are stored at each program node. To construct the path graph, we start at the node where a vulnerable or don't-know resolution was derived. These nodes are first added to the graph as entry nodes. Marple then determines for each successor, whether the query at the current node was actually propagated from either of its successors; if so, the successor(s) is added to the graph, and an edge between the predecessor and successor is also added into the graph. The process continues until the suspicious statement, where the query was initially raised, is reached.

In testing, generating an input that potentially covers a path segment in a path graph is more efficient than generating an input based on individual path segments. The reasons are as follows. Concolic testing generates a test input for a new path by inverting a particular branch. Given a path, concolic testing potentially needs to invert a set of branches from an initial execution, and take several iterations before a desired test input can be generated. On the other hand, if a set of path segments are given in a graph, concolic testing has more flexibility in choosing which path to exploit. The testing terminates as long as any suspicious path segment in the graph is triggered.

It should be noted that by integrating suspicious path segments into a path graph, some path-sensitive information is potentially lost; that is, a path segment found in the path graph might not be a suspicious path segment reported by Marple. To remove this imprecision, we can annotate path identifications (e.g., using queries) for each edge during the construction of the path graph. The tradeoff is that using the annotated path graph, more information needs to be compared at runtime, incurring additional performance overhead, while the imprecision of a path graph might lead the test input generation to some safe path.

4.3 Reachability Analysis

In our dynamic testing, we need to generate test input for the path that starts at the beginning of the program and

traverses any path segment in path graphs. We use reachability analysis to determine whether any of the branches in a program can actually reach the entries of the path graphs; if not, we terminate the test input generation along the corresponding branch.

Algorithm 1 takes the interprocedural control flow graph of a program (ICFG), and a set of path graphs reported by Marple as inputs. The results of reachability is stored in a map, where for each branch, we report a set of entries of path graphs that the branch can reach. In Algorithm 1, lines *1-5* determine for each branch statement, whether the entries of the path graph can be reached. The core analysis is achieved in a recursive procedure *Reach* (see line *8*). At line *10*, we get the immediate successors of the current branch *b*. For each successor b_i, if b_i is an entry of the path graph, then we add it to the set *reachable* at line *13*; otherwise, we recursively call procedure *Reach* on b_i at line *14*.

Algorithm 1. Calculating Reachability Relationship

```
INPUT: icfg: the ICFG of the program, G: a set of path graphs
OUTPUT: reachability: a map<branch, <set> entries of path graphs>
1   for each branch statement b in icfg
2       initialize reachable {}
3       Reach(b, &reachable)
4       reachability[b] := reachable
5   end
6   return reachability
7
8   PROCEDURE: Reach(statement b, set reachable)
9   //recursively traverse statements can be reached from b
10      set successors := immediate successor statements of b in icfg
11      for each statement b_i in successors
12          if b_i is an entry of any path graph g ∈ G
13              reachable.push(b_i)
14          Reach(b_i, reachable)
15      end
16  end
```

5. Dynamic Testing

In our dynamic testing, we apply the reachability information and the suspicious statements/path segments computed above. Since collecting and solving symbolic constraints are important for generating the test input, we symbolically model fault conditions, as well as the semantics of certain program statements that are relevant to trigger faults. In this section, we first present the goals of program instrumentation and techniques. We then show our modeling techniques for four types of program statements. Finally, we explain how the static information is used to generate test inputs.

5.1 Program Instrumentation

Instrumentation is inserted in the program source. Dynamic testing runs on the instrumented programs and takes actions according to the instrumentation. Four types of actions are applied based on the types of program statements; a general goal is to collect symbolic path constraints and fault conditions needed for test input generation at runtime:

- if an input statement is encountered, we add a new input variable into a *symbolic map*. The symbolic map records the symbolic values of current live variables, and also symbolic path constraints and fault conditions;
- if a binary and unary variable operation is met, we record the symbolic values of the results;
- for conditional branches, we record the conditions for both false and true branches in the symbolic map; and
- for a suspicious statement, such as *strcpy* or pointer dereferences, we construct fault conditions to determine inputs that can trigger the fault.

5.2 Buffer Overflow Vulnerability Model

One main difference of MAGIC and concolic testing is that MAGIC is focused to trigger particular types of fault. Here, we describe the vulnerability model we developed for buffer overflow. In this model, we provide a mapping from a buffer write statement to an overflow condition. We also provide the actions we take at statements related to buffer and pointer operations. For each buffer, we not only consider the buffer size and the string length, but also the contents of the buffer up to a certain number of bytes. We specify a buffer using a 3-tuple (*size*, *L*, *C*), where:

- *size* is a symbolic expression for the size of the buffer;
- *L* = $\{len_1, len_2, ..., len_n\}$ is a set of symbolic expressions representing the lengths of strings stored in the buffer, as shown in Figure 6. Since $'\backslash 0'$ can occur multiple times in a buffer, we record string lengths that are relevant to every $'\backslash 0'$ for precision. Dependent on the location of the pointer p to the buffer, the string obtained via p can be relevant to any of recorded lengths.
- *C* = $c_1, c_2, ..., c_v$ is a sequence of symbolic expressions representing the first *v* characters of the buffer [18]. The tradeoff here is that the more content of a buffer is modeled, more precise the symbolic analysis can achieve, however, with higher overhead.

We use a pair (*addr*, *off*) to specify a pointer to a buffer, where *addr* is the beginning of the buffer and *off* specifies the symbolic offset from *addr*. In Table 1, we show how buffer overflow conditions can be constructed based on the type of program statements at runtime. In the first column, we show three examples of suspicious statements. We explain the construction of buffer overflow conditions in the second column. Consider the first row of the table as an example. p_s and p_d are two pointers. A string in the buffer

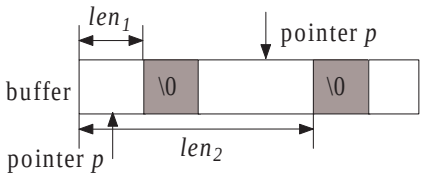

Figure 6. Multiple Strings in a Buffer

Table 1. Modeling Buffer Overflow Conditions

Suspicious Operations	Overflow Conditions
Suppose $p_d = (addr_d, off_d)$ Suppose $p_s = (addr_s, off_s)$ *strcpy*(p_d, p_s)	(b_d ($size_d$, L_d, C_s) := $\delta(addr_d)) \neq null$; ($b_s$ ($size_s$, L_s, C_s) := $\delta(addr_s)) \neq null$; $Len(b_s, p_s)$ - off_s >= $size_d$ - off_d
Suppose $p_d = (addr_d, off_d)$ Suppose $p_s = (addr_s, off_s)$ *strcat*(p_d, p_s)	(b_d ($size_d$, L_d, C_s) := $\delta(addr_d)) \neq null$; ($b_s$ ($size_s$, L_s, C_s) := $\delta(addr_s)) \neq null$; $Len(b_s, p_s)$ - off_s + $Len(b_d, p_d)$ >= $size_d$
Suppose *p* = (*addr*, *off*) **p* := *var*	(*b* (*size*, *L*, *C*) := δ(*addr*)) $\neq$ *null*; *off* > *size*

pointed to by p_s is copied to the buffer pointed to by p_d. At runtime, when such a suspicious statement is encountered, we first find in the symbolic map the buffers associated with p_s and p_d. This action is specified using δ in the second column. If the mapping is successful, shown as $b_d \neq null$ and $b_s \neq null$ in the table, we determine whether the string from p_s copied to p_d potentially cause an overflow. See Figure 7.

In the figure, we show that the available buffer space is $[size_d - off_d]$. The string being copied has a length of $[len_s - off_s]$, where len_s represents the length of the string stored in buffer s. As we mentioned in Figure 6, multiple string lengths can be recorded for a buffer, and we need to select a proper length depending on the location of the pointer. We use $Len(b_s, p_s)$ to represent this action. The second column first row in the table indicates that if the string length is larger than or equal to the available buffer size, a buffer overflow can occur.

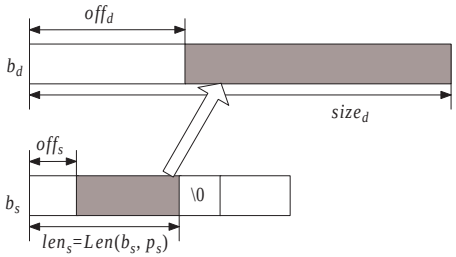

Figure 7. Buffer Overflow Condition

Besides operations modeled by concolic testing [16], our testing components also model additional buffer and pointer operations. Table 2 presents a partial list. In the first column, we present the type of program statements, and in the second column, we specify actions MAGIC takes at the statement to construct the symbolic map. Consider the first row of the table. When the program executes the statement *p* (*addr*, *off*) := *input*(*size*), MAGIC creates a new buffer *b* on the symbolic map. The three parameters of a buffer are initialized: the size of the buffer is *size*, a string length is also *size*, and the first *n* bytes of characters are set based on the input string. After the buffer is created, MAGIC establishes a mapping between the buffer and the pointer using *addr:=&b* and *off:=0*.

Table 2. Symbolic Semantics of String and Pointer Operations

Operations	Symbolic Semantics
p (*addr*, *off*) := *input(size)*	create a buffer *b* (*size*, {*size*}, {c_1, c_2, ..., c_n}), *addr* := *&b*, *off* := *0*
p (*addr*, *off*) := *malloc(size)*	create a buffer *b* (*size*, {}, {}), *addr* := *&b*, *off* := *0*
suppose *p* = (*addr*, *off*) *free*(*p*)	*b* := δ(*addr*) delete *b*
suppose *str* = (*addr*, *off*) *char str*[*size*]	create a new buffer *b* (*size*, {*size*}, {}), *addr* := *&b*, *off* := *0*
suppose $p_d = (addr_d, off_d)$ suppose $p_s = (addr_s, off_s)$ $p_d := p_s \pm v$	$add_d := add_s$, $off_d = off_s \pm v$
suppose *p* = (*addr*, *off*) **p* := '\0'	*b* (*size*, *L*, *C*) := δ(*addr*) *L* := *L* $\cup$ {*off*}

5.3 Path-Guided Test Input Generation

Algorithm 2 takes the path graphs of suspicious path segments reported by Marple, and generates the program inputs that can trigger faults.

At line *2*, initial program inputs are generated. Primitive input variables are given value *0*, and strings are initialized as empty strings. If boundary values of the inputs are known, MAGIC uses their lower and upper bounds.

In the second step, MAGIC executes the program with the generated inputs (see *Run_Program* at line *3*). During execution, MAGIC collects both the branches that the execution covers and the symbolic path constraints at the branches. When a path is discovered to traverse a suspicious path segment in any given path graph, MAGIC determines if a buffer overflow occurs; if not, MAGIC integrates the buffer overflow conditions with a set of path constraints and generates a new test input. If a buffer overflow is confirmed, MAGIC removes the path graph corresponding to this suspicious statement. Also, when a buffer overflow is determined, MAGIC allocates a new memory space for the buffer with the overflowed size and continues the execution, in an attempt to trigger more faults along the same execution. When the execution terminates, we store a sequence of branches and symbolic path constraints, collected along the execution, into the branch list *B* and the constraint list *C*.

Path_Guided_Search at line *6* uses the branch list *B* and the constraint list *C* collected from the previous execution to generate test inputs that excise the suspicious paths. The *for* loop at line *8* examines the collected branch one by one in a reverse order; for each branch, MAGIC determines whether its *paired branch* is either able to reach the entries of any path graph in G (see line *11*) or on a suspicious path segment (see line *15*). If so, at line *21*, MAGIC attempts to generate a new test input using a set of path constraints collected along $\langle B[1], B[2], ..., B[i-1] \rangle$ with the inverted path constraint at branch $\overline{B[i]}$. The generated input at line *23* executes path $\left\langle B[1], B[2], ..., B[i-1], \overline{B[i]} \right\rangle$. If the current branch cannot reach the entries of the path graphs or is not on any suspicious path segment, MAGIC proceeds to examine the next branch at line *8* in the same way. The testing process terminates when all suspicious statements are triggered or all suspicious path segments are covered.

Algorithm 2. Path-Guided Test Input Generation

```
INPUT: G{g1, g2, ..., gl}: a set of path graphs of suspicious path segments
1   initialize branch list B {} and constraint list C {}
2   I = GenInitInput()
3   Run_Program(I, &B, &C)
4   Path_Guided_Search(B, C)
5
6   PROCEDURE: Path_Guided_Search(branchlist B, constraintlist C)
7       bool inversePath := false
8       for(i:=sizeOf(B); i >= 1 ; i- -)
9           get the PairedBranch B[i]
10          for each suspicious path graph gk in G
11              if B[i] can reach any stop point in gk
12                  inversePath := true
13                  break
14              end
15              else if path⟨B[1], …B[i-1], B[i]⟩ traverses a path segment in gk
16                  inversePath := true
17                  break
18              end
19          end
20          if(inversePath = true)
21              I' := solve (C[1] ∩ C[2] ∩ ... ∩ C[i-1] ∩ ¬ (C[i]))
22              initialize branch list B' {} and constraint list C' {}
23              Run_Program(I', &B', &C')
24              Path_Guided_Search(B', C')
25          end
26      end
27  end
```

6. Implementation and Evaluation

We implemented MAGIC for testing buffer overflows. Our goals are to evaluate its capability for generating inputs to detect and trigger faults and also to determine its performance. For comparison, we also constructed two other tools. Tool I implements the techniques of SPLAT [18], which model buffer lengths and the first several bytes of buffer content on top of basic concolic testing to trigger buffer overflows. Different from MAGIC, it does not use boundary values as initial inputs, and terminates when a fault is found. In the experiments, we needed to fix the fault and run the tool again until no more faults were found. We constructed Tool II by isolating the dynamic testing components from MAGIC; that is, it does not use any static information. Comparing Tool II and MAGIC, we can determine the usefulness of the path information in guiding test input generation.

MAGIC is implemented on top of CREST [3] and Marple [13]. Both MAGIC and Tool I are implemented using Microsoft Phoenix SDK[1], and applied the Yices constraint solver [2]. The machine used for experiments is the Intel Duo Core 2.26 GHz processor with 2GB memory. We selected a set of benchmark programs, including WuFTP-1, Sendmail-2, polymorph-0.4.0 and gzip-1.2.4. The first two are buffer overflow benchmarks [20], containing typical and realistic buffer overflows. Polymorph-0.4.0[3] is a real-world program, used to simplify file names in UNIX. Gzip-1.2.4[4] is a file compression program.

We ran a preliminary set of experiments to determine the time threshold we could use for the tools. The experimental results show that for all of the benchmarks, testing either terminates within *1500* seconds, or is no longer able to trigger more faults beyond *1500* seconds. Thus, in our experiments, we decided to double the number and set the time threshold at *3000* seconds. The goal is to ensure that for most of the benchmarks, testing terminates before reaching this threshold, and even when the termination is forced by the time threshold, the number of faults reported by the tools reflect the actual detection capability of the tools.

6.1. Capability of Triggering Faults

We first ran experiments to determine the capability of the three tools for triggering faults. The results are shown in Table 3. The first two columns give the benchmark programs and their sizes. For each tool, we show the number of faults triggered, the number of faults that were missed and the total time that it takes to finish the testing. By manually confirming suspicious statements/path segments reported from Marple, we are able to know the number of buffer overflows a testing tool is supposed to trigger. We therefore can determine the number of faults missed in testing. Also, here we only report the performance of dynamic testing. The overhead of static analysis can be found in the Marple paper [13].

Comparing the results of Tool I and Tool II, we find that more faults are triggered using Tool II than Tool I. Across all benchmarks, Tool I missed *10* faults and Tool II only missed *2*. The reasons for being able to trigger more faults in Tool II are: 1) MAGIC models string contents more carefully, e.g., tracking multiple $'\backslash 0'$ for a buffer; and 2)

[1] http://connect.microsoft.com/Phoenix
[2] http://yices.csl.sri.com/
[3] http://sourceforge.net/projects/polymorph/
[4] http://www.gzip.org/

Table 3. Comparison of Testing Time and Fault Detection Capability

Benchmarks	Size (LOC)	Tool I: SPLAT techniques			Tool II: MAGIC without Static Information			MAGIC		
		Detected	Missed	Time	Detected	Missed	Time	Detected	Missed	Time
WuFTP-1	0.4k	2	0	1342 s	5	0	1325 s	5	0	20 s
Sendmail-2	0.7k	3	1	1618 s*	4	0	1459 s	4	0	171 s
polymorph-0.4.0	1.7k	0	7	>3000 s	5	2	>3000 s	5	2	>3000 s
gzip-1.2.4	8.2k	3	2	463 s	5	0	1071 s	5	0	951 s

* Constraints solver crashed when applying SPLAT on s2, this is the time when constraints solver crashed.

Table 4. Comparison of Test Input Generation Costs

Benchmarks	Tool I: SPLAT techniques			Tool II: MAGIC without Static Information			MAGIC		
	Attempts	Generated	Time	Attempts	Generated	Time	Attempts	Generated	Time
WuFTP-1	13995	1748	30897ms	7828	1254	19408ms	23	20	199ms
Sendmail-2	1335	1084	54531ms	30377	1201	3869ms	5362	253	685ms
polymorph-0.4.0	492061	3335	116743ms	46019	1615	66658ms	227	122	690ms
gzip-1.2.4	4258	485	8652ms	12533	1178	25628ms	5687	1178	5005ms

MAGIC uses boundary values, instead of a fixed default value, which enables more branches to be covered in testing. The times used in testing are comparable for the two tools, except for gzip-1.2.4, where Tool II executes more paths than Tool I due to the use of the boundary value, and thus takes longer to terminate. Since more paths are executed, more faults are found.

Comparing Tool II to MAGIC, we discover that 1) both the tools trigger the same number of faults, which shows Marple does not report false negatives that can impact this testing, and 2) MAGIC is more efficient to find these faults. The testing time is reduced because paths which do not traverse any suspicious path segment are avoided. Among the benchmarks, the time reduction in gzip-1.2.4 is the least. One reason is that for this benchmark, Tool II is not able to cover a certain number of paths due to complex symbolic constraints, and thus testing terminates early. Another reason is that for this benchmark, some of the don't-know path segments are short, and thus in MAGIC, the guidance is not significant.

6.2. The Effort to Generate Test Inputs

In another experiment, we compared the effort of generating test inputs with the three tools. Table 4 presents the experimental results for each tool. Under *Attempts*, we display the number of paths (or path segments) that are targeted for test input generation, i.e., the number of times that symbolic constraints are sent to the constraint solver for potential test inputs. Under *Generated*, we give the number of test inputs that are successfully generated from the constraint solver. The numbers count both the test inputs that can trigger faults, and the inputs generated in the process of searching for suspicious path segments. Under *Time*, we show the total time spent in the constraint solver in generating test inputs from the symbolic constraints.

Our experimental results show that MAGIC largely reduces the search space for generating test inputs, as both the number of paths explored and the number of test inputs generated in the testing process reported by MAGIC are much less. The time used for test input generation is also reduced accordingly.

7. Related Work

There is a large body of work on detecting software faults statically and dynamically. Most of the dynamic methods, such as ProPolice [10] and CRED [15], focus on how to instrument code and detect software faults more efficiently at runtime, rather than generating test inputs to trigger vulnerabilities. In this section, we only include the work that is closely related to our research, including path based test input generation, and hybrid solutions for faults.

Techniques for path based test input generation have three general categories: 1) EXE [5] and KLEE [4] symbolically execute a program along program paths, and generate inputs based on collected symbolic path constraints; 2) SAGE applies trace information and symbolic path constraints to generate test inputs [11]; and 3) DART [12], CUTE [16], SPLAT [18] and CREST [3] are concolic testing tools, which use both symbolic and concrete values to generate the test inputs. CREST [3] proposes several search strategies to improve the branch coverage for concolic testing. SPLAT [18] models buffer operations and determines at runtime whether a buffer overflow can occur at each buffer access; if so, SPLAT generates a test input to trigger the fault. The above techniques all exhaustively explore program paths to generate test inputs, and thus the scalability is an issue. Testing has to give up when a certain number of paths are executed. MAGIC is different in that the testing is guided along suspicious paths, and it explores the paths that are more likely to trigger the faults.

Another area of related work includes hybrid techniques for software faults. Aggarwal et al. [1] integrate static information computed by BOON [17] to reduce the number of test inputs for more efficient dynamic analysis. There are

also tools Check 'n' Crash and DSD-Crasher [6] [7] that detect vulnerabilities in Java programs. Check 'n' Crash uses fault conditions computed by ESC/JAVA for generating test inputs that can trigger faults. Since the static information is intraprocedural, Check 'n' Crash reports high false positives. DSD-Crasher applies Daikon [9] to infer predconditions for each function to remove some of the false positives. Different from these tools, MAGIC applies statically computed interprocedural information to guide the testing, and the faults triggered are real faults. Also, path information instead of information at a program point is used, making the test input generation more cost-effective.

8. Conclusions

This paper presents MAGIC, a path-guided concolic testing framework for automatically generating test inputs to exploit statically identified faults. MAGIC consists of both the static and dynamic components: the static components include a path-sensitive analyzer, and a reachability analyzer; and dynamic components implement concolic testing that in particular is able to trigger buffer overflows in a program. Our experiments show that in MAGIC, the dynamic testing confirms statically reported buffer overflows, and also determines some of the don't-know static warnings as faulty. MAGIC also helps classify false positives, as if no inputs can be generated to exploit the suspicious path, we are more confident that the suspicious path is safe. We also find that guided by the path information, our testing runs much faster than concolic testing in identifying all of the faults. Although we only implemented buffer overflow detection for our experiments, more types of faults can be included.

References

[1] A. Aggarwal and P. Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference*, pages 343–350, 2006.

[2] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, 2007.

[3] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, 2008.

[4] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224, 2008.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, 2006.

[6] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 422–431, 2005.

[7] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–37, 2008.

[8] P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electron. Notes Theor. Comput. Sci.*, 217:5–21, 2008.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 213–224, 1999.

[10] H. Etoh. Gcc extension for protecting applications from stack smashing attacks, December 2003.

[11] L. M. Y. GODEFROID, P. and D. MOLNAR. Automated whitebox fuzz testing. In *NDSS '08: Proceedings of Network and Distributed Systems Security*, 2008.

[12] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[13] W. Le and M. L. Soffa. Marple: a demand-driven path-sensitive buffer overflow detector. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 272–282, 2008.

[14] W. Le and M. L. Soffa. General, scalable path-sensitive fault detection. *Technical Report CS-2010-11 by Computer Science Department, University of Virginia*, 2010.

[15] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed Systme Security Symposium*, pages 159–169, 2004.

[16] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, 2005.

[17] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, February 2000.

[18] R.-G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 27–38, 2008.

[19] M. Young and M. Pezze. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.

[20] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106, 2004.