Marple: Detecting Faults in Path Segments Using Automatically Generated Analyses

Wei Le, Rochester Institute of Technology
Mary Lou Soffa, University of Virginia

Generally, a fault is a property violation at a program point along some execution path. To obtain the path where a fault occurs, we can either run the program or manually identify the execution paths through code inspection. In both of the cases, only a very limited number of execution paths can be examined for a program. This article presents a static framework, Marple, that automatically detects path segments where a fault occurs at a whole program scale. An important contribution of the work is the design of a demand-driven analysis that effectively addresses scalability challenges faced by traditional path-sensitive fault detection. The techniques are made general via a specification language and an algorithm that automatically generates path-based analyses from specifications. The generality is achieved in handling both *data-* and *control-centric* faults as well as both liveness and safety properties, enabling the exploitation of fault interactions for diagnosis and efficiency. Our experimental results demonstrate the effectiveness of our techniques in detecting path-segments of buffer overflows, integer violations, null-pointer dereferences and memory leaks. Because we applied an interprocedural, path-sensitive analysis, our static fault detectors generally report better precision than the tools available for comparison. Our demand-driven analyses are shown to be scalable to deployed applications such as *apache*, *putty* and *ffmpeg*.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Algorithms, Reliability, Security, Experimentation

Additional Key Words and Phrases: Path Segments, Demand-Driven, Faults, Specification

ACM Reference Format:
Le, W., Soffa, Mary Lou, 2011. Marple: Detecting Faults in Path Segments Using Automatically Generated Analyses. ACM Trans. Softw. Eng. Methodol. 0, 0, Article 0 (2011), 34 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The use of static analyses to check for properties in programs, such as protocol violations and security vulnerabilities, has been growing with the recognition of their effectiveness in improving software robustness and security. A main challenge of building and using such static tools in practice is the high cost. First, manual effort has to be invested to diagnose static warnings and exclude false positives [Bush et al. 2000; Chen and Wagner 2002]. Locating root causes and introducing correct fixes can also be difficult. Unlike debugging, often only little information, such as the line number in the source where a fault potential occurs, is provided as static warnings. Furthermore, static fault detectors are often difficult to scale for large software. Manually developed annotations are sometimes inserted to help with the scalability [Evans 1996; Hackett et al. 2006]. Tools, e.g., ESP [Das et al. 2002], can hardcode heuristics targeting one type of fault or even a particular program for

This work is partially supported by Microsoft Phoenix Research Awards.
This article presents new materials as well as integrates two previous conference papers by the authors, published respectively in Proceedings of the 16th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2008), and Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA 2011).
Author's addresses: W. Le, Golisano College of Computing and Information Sciences, Rochester Institute of Technology; M.L. Soffa, Department of Computer Science, University of Virginia.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

precision and scalability. While constructing and tuning a static fault detector is time-consuming, repeated effort has to be invested to build individual analyzers.

At runtime, a fault is developed along an execution path and occurs at a program point. Ideally, to statically predict a potential fault, we need to analyze each execution path and report a sequence of statements along the path that contribute to the production of a fault. The challenge is that there exists a potentially exponential number of paths in a program, and exhaustively enumerating such a large state space is not feasible. Advanced static techniques that target the state space explosion problem face significant shortcomings. Randomly selecting program paths for finding faults [Dwyer et al. 2007] sacrifices the coverage of static analysis, and thus potentially misses faults. Abstracting and merging states among multiple program paths using techniques such as abstract interpretation [PolySpace 2001], model checking [Chen and Wagner 2002], or heuristics based dataflow analysis [Das et al. 2002] can lose precision, causing many false positives. There is also region-based static analysis which summarizes and composes the information from different procedures. This approach is not always able to precisely track faults across procedures [Xie and Aiken 2007; Hallem et al. 2002; Dillig et al. 2011].

This article presents a static, path-based framework, Marple, which addresses the challenges of precision, scalability and generality faced by traditional static fault detection. Our overall goal is a practical tool that can find faults in real-life software with reasonable precision and affordable overhead. The novelty of the framework is a demand-driven algorithm that enables efficient interprocedural, path-sensitive analysis with a desired fault detection coverage. The path segments where the root causes of a fault are located are returned by Marple, providing both focus and necessary information for users to diagnose the static warnings. With the specification techniques we design for users, Marple is able to atomically produce individual static analyses that detect user-specified faults.

Our insight and assumption behind the techniques is that faults manifest locality. In many of the cases, not every program path and every program state are attributable to the production of a fault. Therefore, by directing static analysis to the code that likely contains a fault and reporting path segments that are relevant to the fault, we potentially avoid collecting useless information and achieve both scalable and precise fault detection. In our preliminary study, we have shown that such analysis can scale to at least 570 k lines of code in detecting buffer overflows, and only 43% nodes and 52% procedures are visited during fault detection [Le and Soffa 2008]. We hypothesize that this technique is generally applicable and can be easily extended to different types of faults via modularization.

At a high level, Marple converts the fault detection problem to queries about program facts, such as relations or values of program variables, or orders of procedural calls, at program points of interest. These program points are locations where a fault, i.e., the property violation, is potentially observed at runtime. The query questions whether the required safety constraints can be satisfied at these program points. By propagating and finally resolving the query, we answer whether the paths the query traversed would lead to a fault at the program point. Depending on the types of queries, a backward or forward static analysis is performed, and the information in the program source is collected for symbolically updating the query. The analysis terminates when the resolutions are derived, which indicate the safety of the paths. In summary, Marple is a framework that integrates the following key set of techniques:

(1) a specification language that allows the user to specify a fault and rules for determining the fault;
(2) an interprocedural, path-sensitive, demand-driven, symbolic analysis that resolves integer constraints using the information available in the program source code; and
(3) an algorithm that automatically generates individual static analyzers from specifications.

We implemented the framework and produced the static analyzers for detecting buffer overflows, integer faults, null-pointer dereferences and memory leaks. Our experimental evaluation demonstrates that the demand-driven analysis is feasible and scalable for detecting path segments of faults.

With these efficient algorithms, we are able to achieve interprocedural, path-sensitive precision, producing fewer false positives compared to other static analyzers. With the specification techniques provided with Marple, we avoid the manual effort of repeatedly constructing static analyzers for finding different types of faults. The detection capability and precision of the automatically produced detectors are comparable to the manually tuned ones.

According to our best knowledge, Marple is the first static framework that reports the path segments where different types of faults occur. Compared to a program statement [Xie and Aiken 2007; Xie et al. 2003], path segments contain more information that explains why a fault occurs. Compared to displaying a complete program path [Bush et al. 2000; Das et al. 2002; Microsoft Prefast], highlighting only the relevant path segments provides more focus on where the root causes reside. By applying demand-driven analysis for scalability, Marple is also the first framework that achieves interprocedural, path-sensitive precision without compromising coverage and generality of the fault detection.

In the rest of the article, we first clarify why path information is important for fault detection and diagnosis in Section 2. In Section 3, we give an overview of Marple. The focus is to introduce the components of the framework and their interactions. From Sections 4 to 6, we describe the three key components integrated in the framework, including a specification language, an interprocedural, demand-driven analysis, and an algorithm that automatically generates the fault-specific detectors for user-specified faults. In section 7, we provide our experimental results, followed by a related work in Section 8 and a discussion of the limitations in Section 9. Finally, we conclude our work in Section 10.

2. PATHS FOR FAULT DETECTION AND DIAGNOSIS

To detect a fault statically, we need to define a fault.

DEFINITION 1. *A program fault is a property violation that occurs at a program point during execution.*

A fault is a runtime abnormal condition. When a program runs, a program fault is developed along a sequence of execution; when a particular program point is reached, we observe that the program state at the point does not conform to the property as expected. This abnormal condition can be manifested immediately at the program point, e.g., causing the program to crash, or the corrupted program state can continue to propagate and be manifested later along the execution.

2.1. The Value of Paths

Since the fault is produced after executing a sequence of instructions rather than at a specific instruction, we are not able to statically predict the fault by only matching a syntactic code pattern to each program statement. For the same reason, we should not merge important path information when detecting faults. To achieve a precise fault detection, we need to track the relevant transitions of program states along individual paths to determine if any violation can occur.

Using an example from *sendmail-8.7.5*, we show that false positives can be avoided when we distinguish information from different paths in fault detection. In Figure 1, `strcpy()` at node 5 is not a buffer overflow. A path-insensitive analyzer would merge the facts [`buf = xalloc(i+1), i>=sizeof(buf0)`] from path $\langle 1-3 \rangle$ and [`buf = buf0, i<sizeof(buf0)`] from path $\langle 1, 2, 4 \rangle$, and get the result [`buf = xalloc(i+1)` $\sqcup$ `buf = buf0`] at node 5 (symbol $\sqcup$ represents the union of the two dataflow facts). Since `buf0` is a buffer with a fixed length, and `a.q_user` gets the content from a network package, the analysis identifies node 5 as vulnerable. Whereas, path-sensitive analysis can distinguish that `buf` is set to be `buf0` only along path $\langle 1, 2, 4 \rangle$, while along this path, the length of `a.q_user` is always less than the size of `buf0`, and thus the buffer is safe. In our experiments, our path-sensitive analyzer is aware of the impact of the bounds checking at node 2 and successfully excluded this false positive; however, Splint [Evans 1996], a path-insensitive detector, incorrectly identified it as a fault.

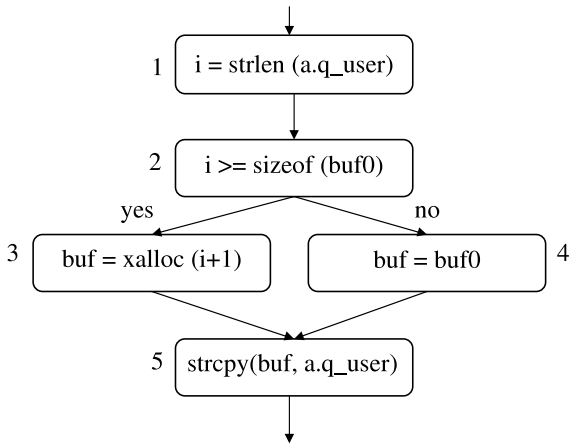


Fig. 1. An Example from *sendmail-8.7.5*: Path-Sensitive Analysis Improves Precision of Fault Detection

Not only does detecting faults need to consider paths, but also reporting a fault should be path-based. Many static tools report faults in terms of the program point where the property violation is perceived. To understand the fault, the code inspector has to manually explore the paths across the point. Among these paths, some might be safe or infeasible, not useful for determining root causes. Even if a faulty path is quickly found, additional root causes may exist along other paths. Without automatically computed path information for guidance, we potentially miss root causes or waste effort exploring useless information, experiencing an ad-hoc diagnostic process.

Consider a code snippet from *wuftpd-2.6.2* in Figure 2. The `strcat()` statement at node 8 contains a buffer overflow; however, among the paths across the statement, path $\langle 1, 2, 4-6, 8\rangle$ is always safe, while path $\langle 1, 2, 4-8\rangle$ is infeasible. Only path $\langle 1, 3-8\rangle$ can overflow the buffer with a $'\backslash 0'$. If a tool only reports node 8 as an overflow, the code inspector may not be able to find this overflow path until the two useless paths are examined.

In the third example, we show that path information also can help correct the faults, as the root cause of a fault can be path-sensitive, i.e., more than one root cause can impact the same faulty statement and be located along different paths. Consider an example from *sendmail-8.7.5* in Figure 3. There exist two root causes that are responsible for the vulnerable `strcpy()` at node 5. First, a user is able to taint the string `login` at node 5 through `pw.pw_name` at node 10, and there is no validation along the path. However, only diagnosing path $\langle 9, 10, 1-5\rangle$ is not sufficient for fixing the bug, as another root cause exists. Due to the loop at nodes $\langle 6, 7\rangle$, the pointer `bp` might already reference an address outside of the buffer `buf` at node 5 with a specially-crafted input for `pw.pw_gecos`. This example shows that diagnosing one path for a fix is not always sufficient to correct the fault. Therefore, paths containing different root causes should be reported.

2.2. Path Classification

Knowing the importance of distinguishing paths in fault detection and diagnosis, we develop a path classification to define the types of paths that are potentially useful. The classification includes the following four categories of paths:

Infeasible: Infeasible paths can never be executed. A path that statically is determined as faulty but actually infeasible is not useful for understanding the root cause or for guiding dynamic tools. Although we are not able to prune all of the infeasible paths in a program statically, research showed that 9–40% of the static branches in a program exhibit correlations [Bodik et al. 1997b], and at

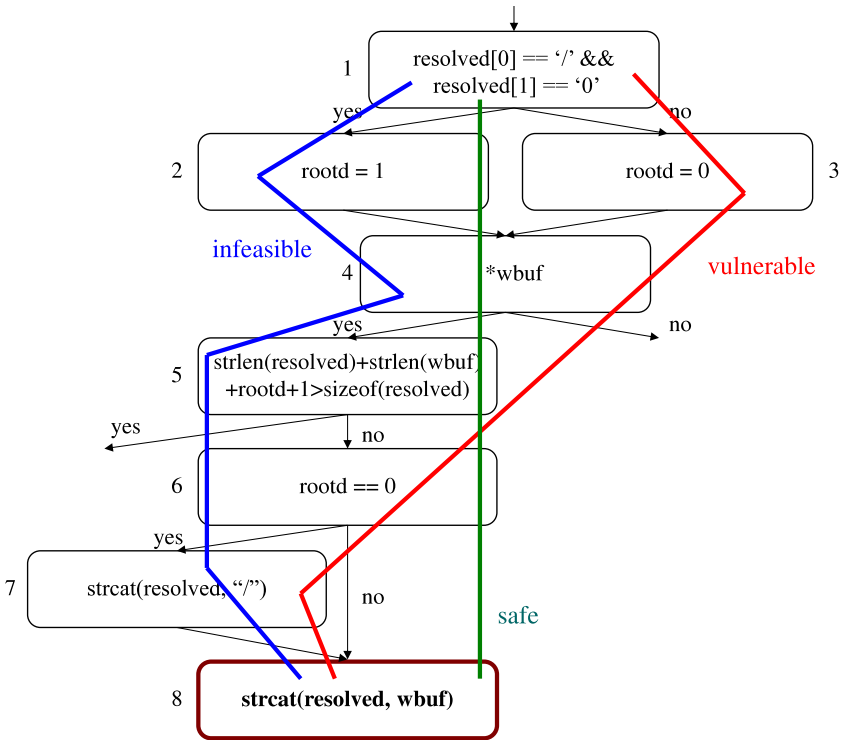


Fig. 2. An Example from *wuftpd-2.6.2*: Different Paths Cross a Buffer Overflow Statement

compile time, we are able to identify infeasible paths caused by these branch correlations. Detecting these infeasible paths statically is important to achieve more precise fault detection.

Safe: Faults are only observable at certain program points, which we call *potentially faulty points*, where violation conditions can be determined. For example, at a buffer access, we can determine whether a buffer overflow occurs. However, not every path that traverses such a program point is faulty. A proper bounds checking inserted before a buffer access can ensure the safetyness of the buffer. Paths that cross a potentially faulty point but do not lead to property violations regardless of the input are called *safe* paths.

Faulty: Executions along a faulty path will trigger the property violation with an appropriate input. Faulty paths can be further classified in terms of the severity of a fault, or the root causes of a fault. In static analysis, we are sometimes able to collect information to predict the severity of a fault, and also can distinguish paths that potentially contain distinct root causes. For example, we determine the severity of a buffer overflow by knowing the source and the contents that are allowed to be written to the buffer. A buffer overflow written by an anonymous network user is certainly more dangerous than the one that only can be accessed by a local administrator. A buffer overflow that can be manipulated by any user supplied contents is more severe than the one written by a constant string. Based on the severity, we can prioritize the buffer overflow warnings reported by static analysis. To distinguish paths with different root causes, we compare statements along a path that contribute to the production of the fault. Two paths with the same sequences of impact statements likely contain the same root causes.

Don't-Know: Besides the above three categories, there are also paths whose safety cannot be determined statically due to the limited power of static analysis, which we call *don't-know* paths. We further classify them based on the factors that cause them to be don't-know paths. The idea is that instead of ad-hoc guessing the values of don't-knows and continuing the analysis with unpredictably

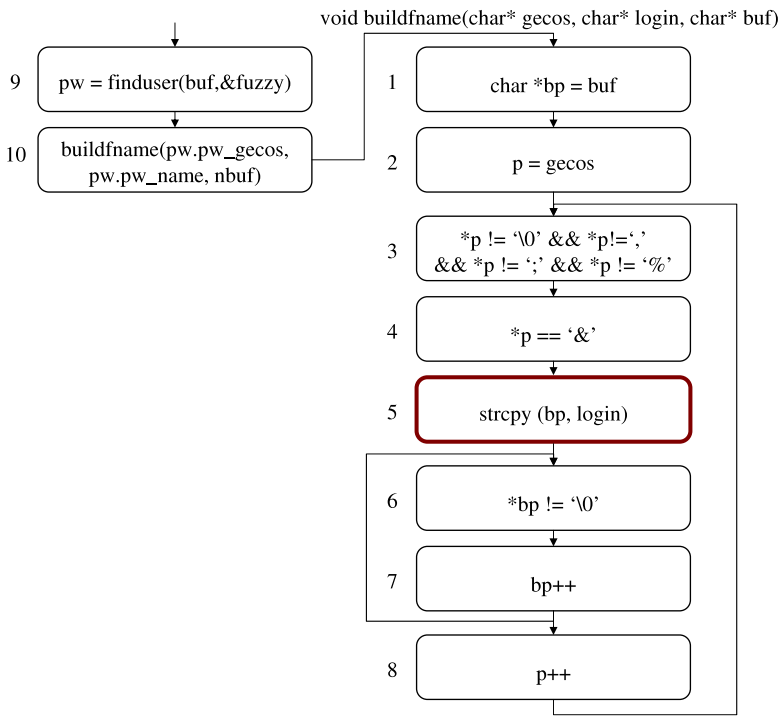


Fig. 3. An Example from *sendmail-8.7.5*: Path-Sensitive Root Causes

imprecise results, we record the locations of these don't-knows as well as the reasons that cause the don't-knows. In this way, code inspectors can be aware of them. Annotations or controlled heuristics can be introduced to further refine the static results if desired. Other techniques such as testing or dynamic analysis can also be applied to address these unknown warnings. As an example, in Figure 4, we show that static analysis cannot determine whether path $\langle 1-6 \rangle$ contains a buffer overflow unless the semantics of a library call `strchr` at node 1 is notified, and we consider such paths as don't-know.

2.3. Empirical Results that Demonstrate the Value of Path-Sensitivity

Applying the path-sensitive buffer overflow detector we developed [Le and Soffa 2008], we performed an empirical study on the value of paths for fault detection and diagnosis. We chose a benchmark suite, consisting of 8 programs, including ones from BugBench [Lu et al. 2005] and the Buffer Overflow Benchmark [Zitser et al. 2004], as well as a Microsoft online XBox game, *MechCommander2*.

2.3.1. Paths for Reducing False Positives. In the first experiment, we compared our path-sensitive buffer overflow detector with Splint [Evans 1996], the only path-insensitive static buffer overflow detector we found that can run through most of our benchmarks and report relatively reasonable false positives and false negatives. We ran our analysis and Splint over the same set of benchmarks. We examined the buffer overflow statements reported by Splint against statements that our analysis identified as containing paths of buffer overflow and don't-know.

In Figure 5, we give detailed comparison for each benchmark program using a set of Venn diagrams. For programs of *polymorph-0.4.0*, *BIND* and *MechCommander2*, Splint terminates with

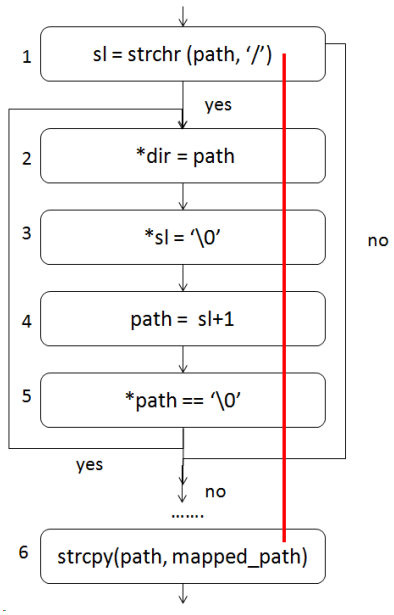


Fig. 4. An Example from *wuftp-2.5*: A Don't-know Path Caused by a String Library Call

parse errors, so we are not able to report a comparison. In the left corner table in Figure 5, under T, we report the total number of statements our analysis reports as containing *vulnerable* (overflow with an external input) , *overflow-input-independent* (overflow with a constant) and don't-know paths. Under T_s, we present the total number of warnings Splint generates for buffer overflow. Comparing these two columns, we discovered that even if we do not use any further techniques to remove don't-knows, our analysis generated less warnings, except for the benchmark *ncompress - 4.2.4*. Splint reported 10 less warnings than our detection on this benchmark because it missed 11 statements we confirmed as overflow.

For each benchmark, we produce two Venn diagrams to show the intersections of results produced by our analysis and Splint. The first Venn diagram displays $(V \cup O) \cap T_s$, the intersection of statements containing paths of overflow-input-independent and vulnerable reported from our analysis and the overflow messages generated by Splint. Set B contains the number of statements that both tools report. Set A reports the total number of confirmed overflows generated by our analysis but missed by Splint. C reports the total number of warnings Splint generated but are not reported by our tool. The second Venn diagram $U \cap T_s'$ compares our don't-know set, U, with the warning set Splint produced excluding the confirmed overflows, annotated as T_s'. In the Venn diagrams, A gives the number of statements listed in U but not reported by T_s'. B counts the elements in both sets. C reports the number of statements from T_s', but not in U.

We summarized the comparison in Figure 6. The diagram shows that for the 5 programs that are successfully compared, Splint and our analysis identified a total of 17 common overflows (see set A in Figure 6), and our analysis detected 19 more flaws that Splint is not able to report (B). There are 38 warnings both reported by Splint and the don't-know set from our analysis (C), and thus those statements are likely to be an overflow. There are a total of 202 warnings generated by Splint but not included in our conservative don't-know set (D). We manually diagnosed some of these warnings including all sets from *ncompress-4.2.4* and *sendmail2*, 10 from *gzip-1.2.4* and 10 from *bc-1.06* that belong to D; we found that all of these inspected warnings are false positives. The number of statements that are in our don't-know set but not reported by Splint is 99 (E), which suggests that Splint either ignored or applied heuristics to process certain don't-know elements in the program.

Benchmark	T	T_s
ncompress-4.2.4	24	14
gzip-1.2.4	21	95
bc-1.06	110	133
wuftp1	6	6
sendmail2	6	8

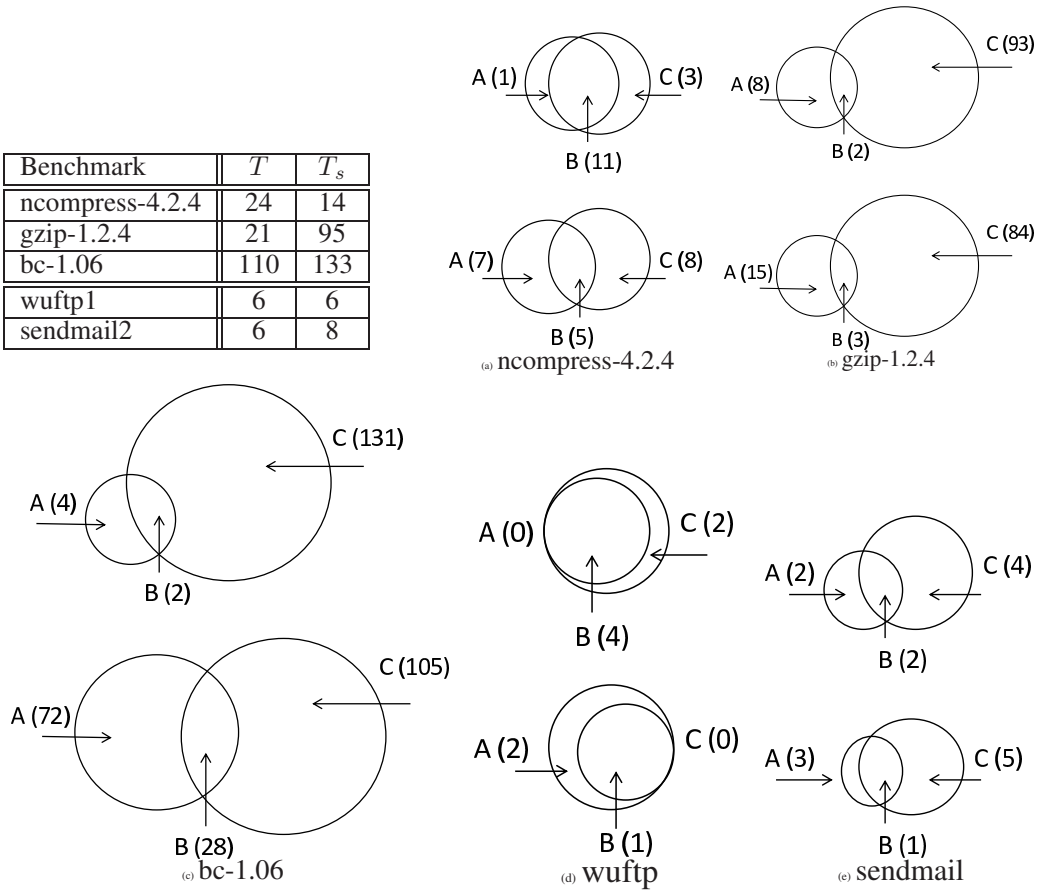


Fig. 5. Comparing Results from Splint and our Analysis: Venn diagrams for $(V \cup O) \cap T_s$ and $U \cap T_s'$

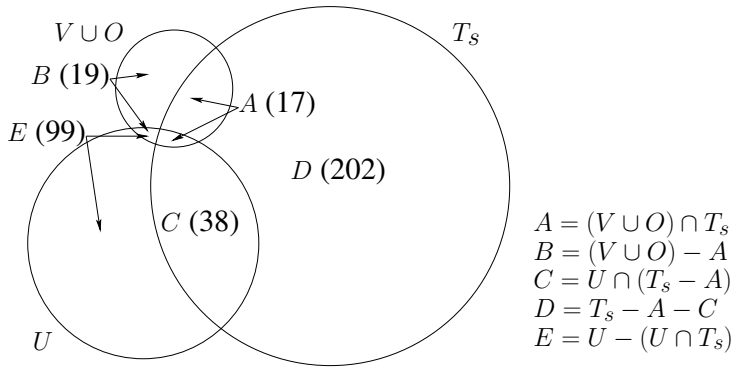


Fig. 6. Summary of Comparison

2.3.2. Paths for Guiding Fault Diagnosis. The goal of our second experiment is to demonstrate that the path classifications do help prioritize faults and can guide fault diagnosis. In this experiment, we collected path classification for detected buffer overflows. The results are given in Table I.

Table I. Detection Capability and Precision of our analysis

Benchmark	Size	Detected Overflow Stmt	Path Prioritization			Root Cause No.
			V	O	U	
polymorph-0.4.0	0.9 k	7	6	1	2	1.7
ncompress-4.2.4	2.0 k	12	8	4	12	1.0
gzip-1.2.4	5.1 k	10	7	3	18	1.7
bc-1.06	17.7 k	6	3	3	108	1.0
wuftp1	0.2 k	4	3	1	4	1.0
sendmail2	0.9 k	4	3	1	6	1.2
BIND	1.3 k	0	0	0	22	N/A
MechCommander2	570.9 k	28	28	0	487	1.0

Under *Detected Overflow Stmt*, we list the total number of overflow statements we detected and manually confirmed. Under V, we report the number of statements that contain vulnerable paths. Under O, we give the number of statements that contain overflow-input-independent paths, and under U, we show the number of statements that contain don't-know paths. We find that paths of different types traverse a buffer overflow statement, and by classifying the paths, we can prioritize vulnerable buffer overflow over overflow-input-independent and don't-know warnings. We also can target don't-know warnings for further refinement.

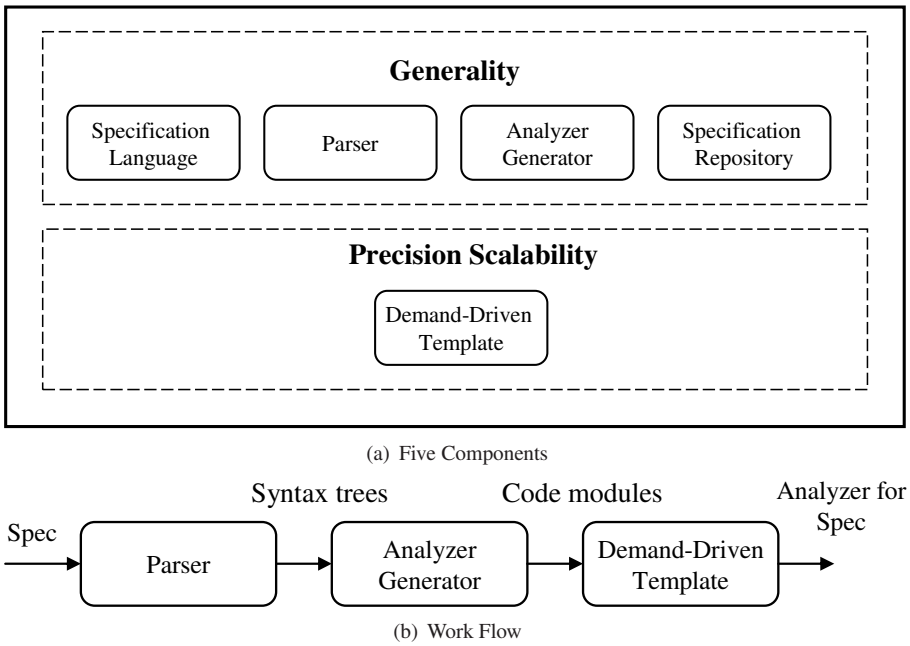
Under *Root Cause No.*, we show the average number of root causes per overflow for all overflow statements in the program. If the result is larger than 1, there must exist some overflow in the program associated with more than one root cause. In our results, we count the number of overflows as the number of statements where the buffer safety constraints are potentially violated, independent on the number of overflow paths that traverse the statements. We count the number of root causes as the number of factors and locations we need to consider in order to correctly fix an overflow. We manually inspected overflow paths and discovered 3 out of 8 programs containing overflows with more than 1 root cause. Interestingly, the different root causes for the overflow are all located on different paths.

3. THE MARPLE FRAMEWORK

In the previous section, we have shown that it is important to identify path information for precise fault detection and also for prioritizing and guiding the diagnostic tasks. In the rest of the article, we present our techniques to automatically detect such path information.

In Figure 7, we give an overview of the Marple framework. Marple takes a user-provided specification, and generates interprocedural, path-sensitive analyses that detect faulty path segments for the specified faults. Marple internally contains five components, shown in Figure 7(a). The *Specification Language* consists of syntax and semantics used to describe faults and fault detection. The *Parser* and the *Analyzer Generator* translate the specification and produce the parts of the analysis that target the specified faults. A general, path-sensitive, demand-driven algorithm is developed through the *Demand-Driven Template*, which implements our design decisions for handling the challenges of precision and scalability of the analysis. The *Specification Repository* consists of specifications we constructed for a set of common faults, including buffer overflows, integer violations, null-pointer dereferences and memory leaks. Users thus can directly run our framework to detect these types of faults. The user also can define her own faults using the language provided by Marple.

As shown in Figure 7(b), given a specification, the *Parser* first produces a set of syntax trees. The *Analyzer Generator* walks the syntax tree, and, based on the semantics, generates the code modules that implement the rules for determining the specified faults. The code modules are plugged into the *Demand-Drive Template* to produce the analyzer. The analyzer takes program source code as input and identify four types of paths defined in Section 2. The specifications for multiple types of faults can be integrated to generate an analysis that handles a set of types of faults. The advantage of such an analysis is that we can reuse the intermediate results—e.g., feasibility or aliasing information, for



(a) Five Components

(b) Work Flow

Fig. 7. The Marple Framework

analyzing different types of faults—and also explore the interactions of different types of faults [Le and Soffa 2010].

4. FADE: A SPECIFICATION LANGUAGE FOR FAULTS

The first component of Marple is the *Specification Language*, which we call *Fade* (FAult DEfinition language). In this section, we first explain how we model faults and their detection. We then give specification examples and the grammar of the language.

4.1. Modeling Faults

In order to statically identify runtime faults, we define a *fault signature* for users to specify a fault and a *detection signature* for expressing information needed to detect the fault. A fault signature consists of two elements: a program point and properties that must hold at the point. We specify program points, using *code signatures* of particular types of statements, and properties, using *constraints*. The constraints define conditions on program states. The key elements to compose constraints are *attributes*. Attributes represent an abstraction of program states and specify properties of program objects such as program variables or statements. For instance, an attribute can be a value, range, or typestate [Strom and Yemini 1986] of individual program variables or relations of multiple variables.

As an example, a buffer overflow occurs at a buffer access when the length of the string stored in the buffer is larger than the buffer size. To model the fault, we identify the code signatures of buffer read and write, and we define the relation of the string length and buffer size as constraints. Similarly, to model "an opened file has to be closed," we find code signatures of "file-open" and construct a constraint as "a file-close has to follow the file-open."

Besides the above examples, we later show that our technique can also model traditional faults, including integer truncation/signedess conversion problems, integer overflows/underflows, null-pointer dereferences and memory leaks. In our model, the constraints can be about the order of operations, which we call *control-centric*, or *data-centric* if the constraints define relations of value

Vars		*Vbuffer* a, b; *Vint* d; *Vany* e;
Fault		
	CodeSignature	$`strcpy(a,b)`$ // when copy b to a
	S_Constraint	Size(a)≥Len(b) // check if size of a is no less than length of b
or		
	CodeSignature	$`memcpy(a,b,d)`$
	S_Constraint	Size(a)≥min(Len(b), Value(d)) // string in a is length of b or d
or		
	CodeSignature	$`a[d]=e`$ // write to location d of a
	S_Constraint	Size(a)>Value(d) // d should not be out of a's boundary
Detection		
	CodeSignature	$`strcpy(a,b)`$ // after b copied to a
	Update	Len(a):= Len(b) // length of a is equal to length of b
or		
	CodeSignature	$`strcat(a,b)`$
	Update	Len(a):= Len(a)+Len(b)
or		
	CodeSignature	$`a[d]=e`$ && Value(e)='\0' //'\0' written in a[d]
	Update	(Len(a)>Value(d) ‖ Len(a)=undef //when no '\0' before a[d], ↦ Len(a):= Value(d) //length of a is d
or		
	CodeSignature	$`d=strlen(b)`$
	Update	Value(d):= Len(b) //semantics of strlen library call

Fig. 8. Partial Specification for Detecting Buffer Overflows (we use '//' to add comments for specifications)

or value range of program variables. The types of faults we are able to model are those that can be defined using a set of constraints whose violations can be observed at certain program points.

To determine the detection signature, we need to understand how a fault is produced dynamically. At runtime, a sequence of changes of program states along the execution path lead to the violation of the property constraints. Therefore, to statically determine the violation of constraints, we need to provide in detection signatures the types of program points that potentially contribute to the production of a fault and their corresponding impact.

The constraints at a program point can express the history or future of an execution, which we call *safety* or *liveness* constraints respectively. For example, determining buffer overflow requires resolving a safety constraint because it is the values generated along the execution path before reaching the buffer access that contribute to the buffer overflow. On the other hand, in the file-open-close example, we require that at the "file-open," the "file-close" should occur in the future, a liveness condition. Since static analysis can be either backwards or forwards, we take into consideration whether a constraint is related to safety or liveness properties when determining the direction of the analysis.

4.2. Specification for Detecting Buffer Overflows

A specification consists of a fault signature and a detection signature. The fault signature consists of program points and constraints, and the detection signature consists of program points and rules for updating the constraints at the program points. To express the fault and detection signatures, the key is to specify the constraints and update rules using attributes of program variables. Here, we provide a specification example constructed for detecting buffer overflows, a type of data-centric faults.

In Figure 8, the keyword *Vars* defines variables used in the specification. Under *Fault*, the keyword *CodeSignature* provides a set of program points where the buffer constraints have to be enforced.

The examples include the library calls of `strcpy` and `memcpy` as well as the direct assignment to a buffer. We use *S_Constraint* to indicate that the buffer overflow constraint is a safety constraint. The constraints can be specified using a comparator "$\geq$" on attributes of *Size(a)*, the size of buffer *a*, and *Len(b)*, the length of the string *b*. In our language, we predefined a set of attributes, such as *Size* and *Len*, whose semantics are implemented in the Marple framework. Users can directly use these attributes to specify their definitions of faults and rules for detecting faults. The role of variables such as *a* and *b* is to locate the operands in the code signature for constructing constraints.

Under *Detection*, we show a set of program points that potentially affect the buffer size or string length as well as the update rules for these program points. The first pair says that after a `strcpy` is executed, the length of the string stored in the first operand equals the length of the string stored in the second operand. The third pair introduces a conditional command using a symbol $\mapsto$. It says when a '\0' is assigned to the buffer, if the current string in *a* is either longer than *d*, *Len(a)>Value(d)*, or not terminated, *Len(a) = undef*, we can assign the string length of *a* with the value of *b*. It should be noted that Marple integrates a symbolic substitution module to automatically handle integer computation, e.g., using rules *Value(x):=Value(y)* for the integer assignment `x = y`. The detection signature provided in the specification only gives rules that are potentially useful for determining the defined faults. In the case of buffer overflow detection, the rules are about string libraries and their semantics.

4.3. Grammar and Semantics

As shown in the previous example, Fade provides a set of commonly used attributes, as well as the operators *computation*, *comparison*, *composition* and *command*. The attributes take program objects and return an integer, Boolean or set. The semantics of attributes are predefined and implemented as library calls in the Marple framework. Based on the domain of the attributes, the corresponding computation and comparison can be applied. The command operators define common actions for updating a constraint, e.g., symbolic substitution or integration of a condition. Fade also provides facilities to pair code signatures with constraints or code signatures with update rules to compose the fault or detection signature.

The grammar of the language is given in Figure 9. In this grammar, we show how a set of advanced language constructs can be composed from the basic construct of attributes. In the grammar, keywords use bold fonts, and terminal symbols, such as the predefined constants, are italicized.

The first rule in Figure 9 says that a specification consists of three sections. In the first section, we list *specification variables*. The specification variables define a set of program objects of interest, such as statements or operands, that a constraint or a symbolic rule would use. The rule *Var* indicates that a specification variable is defined by a type and a name. Fade supports a set of built-in types, listed in the production *VarType*. The naming convention for each type indicates to which category of program objects the type refers. For example, a specification variable of the type *Vint* refers to a program variable whose type is *int*.

After the definition of variables, the grammar provides the fault signature, *FaultSigList*, and the detection signature, *DetectSigList*. *FaultSigList* consists of pairs of potentially faulty points and property constraints, using the keyword **or** for multiple pairs. Similarly, *DetectSigList* lists pairs of impact points and rules for updating constraints at these points. Sometimes, detecting a new type of fault can reuse the rules constructed for detecting other vulnerabilities. Clause *#include* $<$*ExistentSpec*$>$ specifies such reuse.

The construct *ProgramPoint* provides code signatures or/and conditions to identify the types of program statements of interest. We use keywords **S_Constraint** and **L_Constraint** to distinguish whether the fault is related to a safety or a liveness constraint. The production *Condition* composes constraints from attributes using a set of operators. The basic rule is to connect two *Attributes* with a *Comparator*. A condition is a Boolean. Therefore, a set of Boolean operators can be applied. Symbol [ ] is used to specify the priority of the operators. The construct *Action* specifies the actions that can be taken on constraints with the operators of *:=* for replacement of attributes and $\wedge$ for integrating

Specification → **Vars** VarList **Fault** FaultSigList **Detection** DetectSigList
VarList → Var*
Var → VarType *namelist***;**
VarType → *Vbuffer*|*Vint*|*Vany*|*Vptr*|...
FaultSigList → FaultSigItem ⟨**or** FaultSigItem⟩*
DetectSigList → DetectSigItem ⟨**or** DetectSigItem⟩* |*#include <ExistentSpec>*
FaultSigItem → **CodeSignature** ProgramPoint **S_Constraint** Condition|
CodeSignature ProgramPoint **L_Constraint** Condition
DetectSigItem → **CodeSignature** ProgramPoint **Update** Action
ProgramPoint → $LangSyntax$|Condition|$LangSyntax$&&Condition
Condition → Attribute Comparator Attribute|!Condition|[Condition]|
Condition&&Condition|Condition || Condition
Action → Attribute:=Attribute| ∧ Condition|[Action]|Action&&Action|
Action || Action|Condition ↦ Action
Attribute → PrimitiveAttribute(*variable*, ...)|Constant|!Attribute|¬Attribute|[Attribute,Attribute]|
Attribute ∘ Attribute|Attribute Op Attribute|min(Attribute,Attribute)|[Attribute]
PrimitiveAttribute → *Size*|*Len*|*Value*|*MatchOperand*|*TMax*|*TMin*|*IsEnd*|*TypeState*|...
Constant → *0*|*true*|*false*|...
Comparator →= | ≠ | > | < | ≥ | ≤ | ∈ | ∉
Op → +| − | ∗ | ∪ |∩

Fig. 9. The Grammar of the Specification Language, Fade

conditions. An action can be conditional and only be performed when a certain condition is satisfied, which we use the operator ↦ to specify. *Attributes* used to compose *Condition* and *Action* specify the properties of program objects. In our specification language, we define a set of commonly used primitive attributes as terminals, shown in the *PrimitiveAttribute* production. A set of operators are defined to compose attributes from these primitive attributes (see the productions *Attributes* and *Op*).

4.4. Specification for Detecting Memory Leaks

We next show a specification for determining control-centric faults and a liveness property, memory leaks. Here, the constraint is that a memory allocation is safe only if a free of the memory is invoked in the future. In Figure 10, we use the attribute *TypeState(a)* to record the order of operations performed on the section of memory tracked by *a*. The constraint under *Fault* says that when *TypeState(a)* equals 1, the leak does not occur. Under *Detection*, the first rule indicates that if a *free* is called on the tracked pointer, *TypeState(a)* returns 1, and the program is safe. The code signatures from the second to fourth rules present the cases when the pointer is no longer associated with the memory: either it is reassigned, e.g., at *a=b*, or its scope ends, specified by the attribute *IsEnd*. At these program points, we need to determine whether *a* is the only pointer that points to the tracked memory; if so, a memory leak occurs; otherwise, we remove *a* from the reference set, *Ref(a)* (the

Vars		*Vptr* a, b; *Vint* c
Fault		
	CodeSignature	$a=malloc(c)$ //we need to check if a will be released,
	L_Constraint	TypeState(a) == 1 //i.e., whether the typestate of a can be 1
Detection		
	CodeSignature	$free(a)$ //when free on a
	Update	TypeState(a):= 1 //typestate of a is 1
or		
	CodeSignature	$a=malloc(c)$ //at memory allocation
	Update	|Ref(a)|==0↦Ref(a):= {a} ‖ //track a when met first time
		|Ref(a)|==1↦TypeState(a):=0 ‖ //a is the only pointer, leak found
		|Ref(a)|≠1,0↦Ref(a):=Ref(a)-{a} //remove a from aliasing set
or		
	CodeSignature	$a=b$ //a is reassigned
	Update	|Ref(a)|==1↦TypeState(a):=0 ‖
		|Ref(a)|≠1↦Ref(a):=Ref(a)-{a}
or		
	CodeSignature	IsEnd(a) //scope of a ends
	Update	|Ref(a)|==1↦TypeState(a):=0
		|Ref(a)|≠1↦Ref(a):=Ref(a)-{a}
or		
	CodeSignature	$b=a $ //a, b are aliasing
	Update	Ref(a):=Ref(a)+{b} //add b to aliasing set

Fig. 10. Partial Specification for Detecting Memory Leaks

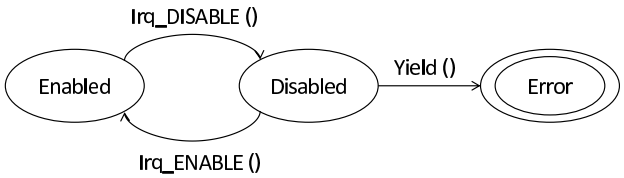
reference set contains a set of pointers that currently point to the tracked memory). The last rule in the specification adds an aliasing pointer to the reference set.

4.5. Constructing Fade Specifications from Finite Automata

Finite automata (FA) have been widely applied to specify control-centric faults [Chen and Shapiro 2004; Das et al. 2002; Chen and Wagner 2002]. Here, we give a case study and show how to convert FAs to our Fade specifications. Intuitively, given an FA, we derive fault signatures from the end states of the FA and the corresponding incoming edges to these end states. Similarly, we obtain the detection signature from transitions between states in FA.

In Figure 11, we show on the left an FA that specifies the property *the program should not call Yield() when interrupts are disabled* [Chen and Shapiro 2004]. A code example that violates this property has been given on the right. Along path $\langle 1-3, 5\rangle$, `Yield()` is called when the interrupt is enabled.

To convert this FA to an Fade specification, we first find the end state, *Error*, and its incoming edge `Yield()`. The two are converted to fault signature, shown in Figure 12, which says when we call `Yield()` in a program, we need to check if the current state of the program enables the interrupt. The control-centric fault here is related to sequence of operations and we use an attribute *State* to track the sequence, Different from memory leak detection, we do not need to track the type states of program variables. To construct the detection signatures, the two transitions in FA are converted. Take the first pair in the detection signature as an example. It says when the call `irq_DISABLE()` is invoked, we transit the state from *Enabled* to *Disabled*. Both in the FA and Fade specification, we did not define what happens when we call `irq_DISABLE()` on a *Disabled* state.




```
1  f(){
2  //interrupt enabled
3    if(condition)
4      g();
5      Yield();
6  }
7  g(){
8    irq_DISABLE();
9  }
```

Fig. 11. the FA for the fault and an Example that Violates the Property [Chen and Shapiro 2004]

Vars		
Fault		
	CodeSignature	$Yield()$ //we need to check when Yield() is called,
	S_Constraint	State == Disabled //whether the interrupt is disabled
Detection		
	CodeSignature	$irq_DISABLE()$ //when irq_DISABLE() is called
	Update	State == Enabled $\mapsto$ State:= Disabled //state turns to Disabled
or		
	CodeSignature	$irq_ENABLE()$ //when irq_ENABLE() is called
	Update	State == Disabled $\mapsto$ State:= Enabled //state turns to Enabled

Fig. 12. Converting FSA to FADE specification

4.6. Usability of the Specification

To specify a type of fault, the user first needs to identify the program points and the constraints that define the fault. If the faults are data-centric, the user can reuse the detection signatures we developed to compute buffer overflows and integer faults. Additional rules also can be introduced to document the semantics of library functions or certain types of operators in the program. If the faults are control-centric, the user needs to identify statements that potentially impact the order of operations defined in the constraint. Users can refer the *PrimitiveAttribute* clause in the grammar to select the predefined attributes for their purposes. To extend Marple for supporting a new type of fault, in the worst case, we need to add a few new primitive attributes and operators. Our assumption is that the required abstractions in the analysis, i.e., attributes, are always limited to certain types, and it is the composition of the attributes that specify different types of faults and their detection.

The size of a specification is dependent on the type of fault we aim to detect, and also the number of statement types we want to include in the fault signature and detection signature. The more statement types we model, the more faults we potentially discover. For example, for detecting buffer overflows, we added 10 pairs of constraints and program points in the fault signature and modeled about 50 different types of program statements, many of which are C libraries related to string manipulations. From our experience, we expect that, for a knowledgeable programmer, a specification may be built within a few hours from scratch for data-centric faults and less than an hour for control-centric faults such as protocol violations.

5. DEMAND-DRIVEN ANALYSIS TO DETECT PATH SEGMENTS OF A FAULT

The second component of the Marple framework is a general, predefined, static analysis template. In the template, we developed an interprocedural, demand-driven, path-sensitive analysis for achieving the desired scalability and precision. The above specifications are then integrated in the template to generate fault-specific detectors.

Demand-driven, in contrast to exhaustive search, refers to a special way in which dataflow is propagated along program paths to collect information for determining faults. To design a demand-driven analysis, the two challenges we need to handle include 1) mapping the problem of fault detection to the queries and their resolutions; and 2) determining the query propagation rules that ensure the path-sensitivity of fault detection. This section provides details of our design for these two aspects. For discussions that are related to code, we use C/C++ as an example.

5.1. An Overview: Detecting Integer Overflows and Null-Pointer Dereferences

We first provide an overview of the analysis by demonstrating a demand-driven algorithm on a simple example, shown in Figure 13. Suppose we aim to detect integer faults and null-pointer dereferences for the given code. In the figure, an integer signedness error occurs at node 11. To detect this fault, the analysis first performs a linear scan and identifies node 11 as a potentially faulty statement, because at node 11, a signedness conversion occurs for integer `x` to match the interface of `malloc`. We raise a query [Value(x)$\geq$0] at node 11, indicating for integer safety, the value of `x` should be non-negative along all paths before the signedness conversion. The query is propagated backwards to determine the satisfaction of the constraint. At node 10, the query is first updated to [Value(x)*8$\geq$0] via a symbolic substitution. Along branch $\langle 8, 7\rangle$, the query encounters a constant assignment and is resolved to $[1024 \geq 0]$ as safe. Along the other branch $\langle 8, 6\rangle$, the analysis derives the information $x \leq -1$ from the false branch, which implies the constraint [Value(x)$\geq$0] is always false. Therefore, we resolve the query as unsafe. The path segment $\langle 6, 8, 10, 11\rangle$ is reported as faulty.

Null-pointer dereferences also can be identified in a similar way. Here, our explanation focuses on how infeasible paths, a main source of imprecision for control-centric faults, are excluded. In our analysis, we first perform an infeasible path detection [Bodik et al. 1997b], and the identified infeasible paths are marked on the interprocedural control flow graph (ICFG), as ip_1 and ip_2 shown in Figure 13. The branch correlation at nodes 2 and 8 regarding variable `test` results in the infeasible path ip_1. Similarly, the value of `flag` at nodes 4 and 12 implies the infeasible path ip_2. To detect the null-pointer dereference, the analysis starts at a pointer dereference discovered at node 13. Query [Value(p)$\neq$NULL] is constructed, meaning the pointer `p` should be non-NULL before the dereference at node 13 for correctness. At branch $\langle 13, 12\rangle$, the query encounters the end of an infeasible path and records ip_1 in progress. Along one path $\langle 13, 12, 9\rangle$, the propagation no longer follows the infeasible path and thus the query drops ip_1. The query is resolved as safe at node 9, assuming memory allocation succeeds and `malloc` returns a non-NULL `p`. Along the other path $\langle 13 - 10\rangle$, no update occurs until the end of ip_2 is met at node 10. The query thus records ip_2 in progress. When the query arrives at branch $\langle 5, 4\rangle$, the start of ip_1 is discovered, showing the query traverses an infeasible path. The analysis thus terminates. Similarly, the propagation halts at branch $\langle 5, 3\rangle$ for traversal of ip_2. The analysis reports node 13 as safe for null-pointer dereference.

5.2. Query and Resolutions

We formulate the fault detection problem to a query system by mapping the constraints of a fault into the queries at certain program points. Take buffer overflow detection as an example. We construct a query at any buffer access regarding buffer *b* and string *s*:

$$[c : \textit{Size(b)} \geq \textit{Len(s)}?]$$

It says that to ensure buffer safety, the program should satisfy the constraint at the buffer access that the size of the buffer, *Size(b)*, should be larger than or equal to the length of the string that intends to be stored in the buffer, *Len(s)*. Similarly, as shown in the previous example, for detecting null-pointer dereferences, we construct a query at the pointer dereference regarding pointer *p*:

$$[c : \textit{Value(p)} \neq \textit{NULL}?]$$

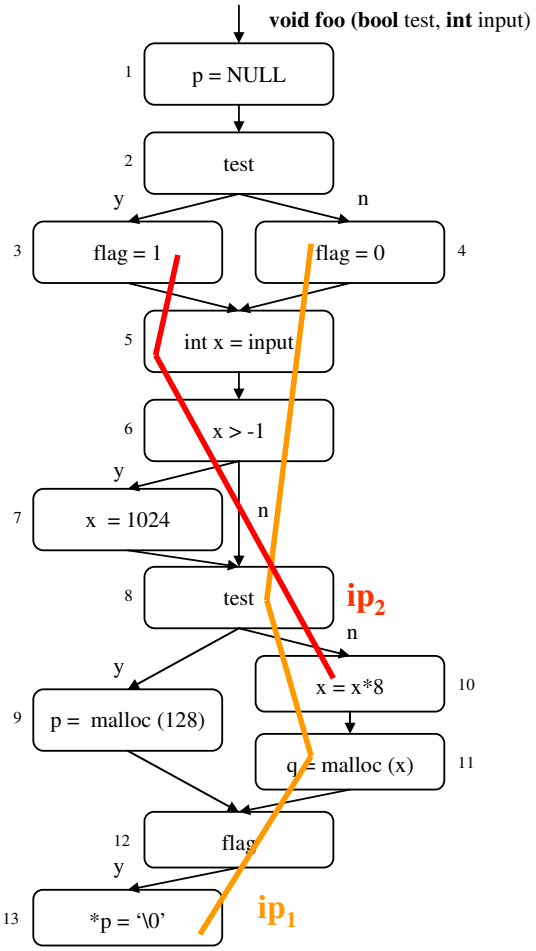


Fig. 13. Detecting Integer Overflow and NULL-pointer Dereference

We also can detect protocol violations such as a file-close only can be invoked on an opened file. In this case, we check whether at the file close, the status of the file is open:

$$[c : \mathit{TypeState(f)} == \mathit{OPEN}?]$$

where *f* is the file under tracking, and *OPEN* is an enum to specify the status of file as "opened".

In addition to safety constraints that specify a fault, the query includes further inquiries about a fault such as the taint information — who can trigger the fault, untrusted users or administers? The query also stores the intermediate analysis results to help with the query propagation, e.g., ip_1 and ip_2 in-progress tracked in Figure 13.

When the conformance of the property constraints in the query is determined, and also information for any additional inquiries is collected, a query is resolved. If the property constraints are determined to be always conformed, no fault is found; otherwise, if violations of the constraints can occur on some external input, a fault exists along the paths to which the query has been propagated. Based on the resolutions, the paths that lead to the statement where a query is constructed are classified as infeasible, safe, faulty and don't-know, and sometimes faulty with different severities.

As traditional static analysis, our query-based, demand-driven algorithm is able to detect faults that can be specified using constraints about program facts, and the program points where the property violations can be observed are identifiable using code signatures.

5.3. Query Propagation: a Path-Sensitive, Demand-Driven Algorithm

To resolve a query, we propagate the query along the program towards where the information for determining the resolutions is potentially located. To track actual execution paths, the query propagation should be interprocedural, path-sensitive, and exclude as many infeasible paths as possible.

Based on the goals, we design a *Demand-Driven Template*, shown in Algorithm 1. The template provides query propagation rules that are generally applicable for identifying different types of faults. The skeleton has "holes", where the fault-dependent information is missing. In Algorithm 1, the "holes" are **MatchFSignature** at line 4 and **MatchDSignature** at line 10. **MatchFSignature** examines whether a given program statement is the program point of interest where a fault potentially is observed; if it is, a query will be constructed using its correspondent safety constraints. **MatchDSignature** determines whether a given statement contains information for updating a query; if so, the query is updated. The two "holes" will be filled in using the code automatically generated from the *Analyzer Generator* in the Marple framework.

ALGORITHM 1: the Demand-Driven Template

Input : program (p)
Output: path segments for faults

```
1  icfg = BuildICFG(p);AnalyzePtr(icfg); IdentifyInfP(icfg);
2  set worklist L to {};
3  foreach s ∈ icfg do
4  |   MatchFSignature(s)
5  |   // hole1: raise query q, if s matched code signature
6  |   if q then add (q,s) to L
7  end
8  while L ≠ ∅ do
9  |   remove (q, s) from L;
10 |   MatchDSignature (q,s);
11 |   //hole2: update query q, if s matched code signature
12 |   a=EvaluateQ (q,s);
13 |   if a ≠Unresolved then add(q,s) to A[q];
14 |   else
15 |   foreach n ∈ Next(s) do PropagateQ(s,n,q);
16 end
17 ReportP(A)
18 Procedure EvaluateQ(query q, stmt n)
19 SimplifyC(q.c, n)
20 if q.c = true then a = Safe
21 else if q.c = false then a = Fault
22 else if q.c = undef ∧ q.unknown ≠ ∅ then a = Don't-Know
23 else a = Unresolved
24 Procedure PropagateQ(stmt i, stmt n, query q)
25  if OnFeasiblePath(i, n, q.ipp) then
26 |   ProcessBranch(i, n, q)
27 |   ProcessProcedure(i, n, q)
28 |   ProcessLoop(i, n, q)
29 end
```

In Algorithm 1, the analysis first builds an ICFG for the program at line 1. The pointer analysis is then performed to determine aliasing information and model C/C++ structures, followed by a branch correlation analysis for detecting infeasible paths. The discovered infeasible paths are marked on the ICFG [Bodik et al. 1997a]. Lines 2–17 of Algorithm 1 present the demand-driven analysis that detects path segments of faults.

The analysis first performs a linear scan of statements in the ICFG to match the fault signature. If the match succeeds, a query will be returned and added to a worklist at line 6.

After the demand is collected, a path-sensitive analysis is performed on the code reachable from where the query is raised. At line 10, if a statement is determined to impact the query, the query will be updated, either via a general symbolic value substitution, provided by the template, or by fault-specific flow functions, supplied via specifications. For each update, we evaluate if the query is resolved at line 12.

Lines 18–23 give details of the query evaluation. At line 19, we simplify the constraints using the algebraic identities and inequality properties. An integer constraint solver is called to further determine the resolution of the constraint. Considering its performance overhead, we do not invoke the constraint solver at every query update but allow users to specify invocation points. For example, the constraint solver can be called whenever a query is propagated out of a procedure. If the constraint returns *true*, the safety rule always can be satisfied; otherwise, if *false*, a fault is discovered. We report the query as *Don't-know* (see line 22) if its resolution is dependent on variables or operations that our analysis cannot handle, e.g., a variable returned from a library or an integer bit operation.

If the query is not resolved, we continue to propagate it for further information. At line 15, **Next** finds the predecessors (in a backward analysis) or successors (in a forward analysis) of the current node. The direction of query propagation is determined by the types of inquiries: if the query requires liveness information, the analysis should be performed forward, and on the other hand, if the query determines a safety property, the analysis should be backward. **PropagateQ** at lines 24–29 integrates a set of propagation rules to handle infeasible paths, branches, procedures and loops, where the path-sensitivity is addressed (see details in Section 5.4).

The analysis terminates when the resolutions for all the queries in the worklist are determined. At line 17, we report path segments which are traversed by the queries. The path segments start where a query is raised and end where the resolution of the query is determined. We take one of the following two approaches to print the results based on the user's request: 1) we can directly list a sequence of nodes the query has been propagated to (this information is stored in the query); or 2) we perform a forward propagation of the query resolutions to the nodes where the query propagated, to obtain all the path segments that correspond to the query and its resolution [Bodik et al. 1997a]. Along a path segment, the constraints of a fault are either 1) always resolved as false, which implies that as long as the execution traverses the path segment, the fault can be triggered independent of the input, or 2) violated only on some user input, which says any execution that crosses the path segments with a proper input can trigger the fault.

To detect different types of faults, code signatures and detection signatures for different types of faults can be combined at lines 4 and 10 respectively in Algorithm 1. Queries for determining the different types of faults can thus be cached and reused at the nodes during the propagation in **PropagateQ**.

5.4. Details for Precision and Scalability

The precision of our analysis is achieved via three orthogonal angles: 1) preserving context-sensitivity, flow-sensitivity, and path-sensitivity on handling control flow; 2) mapping pointers and structures to integer constraints; and 3) handling challenges of symbolic executions. From the three angles, we discuss the tradeoffs we made to balance the precision and scalability of the analysis.

5.4.1. Handling Control Flow. Path-sensitive analysis can differ in precision. One factor is how control flow, such as procedurals, branches and loops, are handled in the analysis. Our path-sensitivity has the following features: 1) we track interprocedural paths [Bodik et al. 1997a] (handled

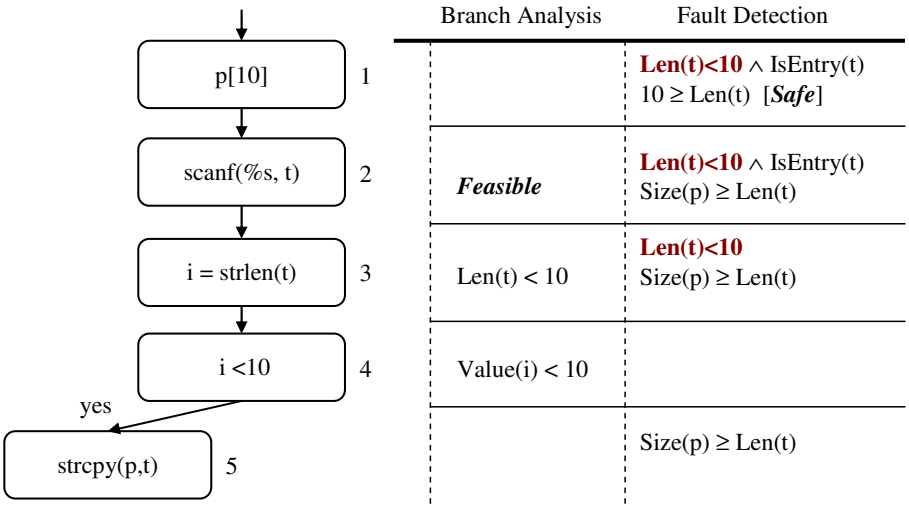


Fig. 14. Reuse the Evaluated Path Conditions

in **Next** at line 15 and **ProcessProcedure** at line 27 in Algorithm 1); 2) queries are not merged at branches and relevant conditional branches are considered (line 26); 3) identifiable infeasible paths are excluded (line 25); and 4) faults are reported as path segments (line 17).

An interprocedural propagation includes two cases. At line 15, if s is the beginning of the procedure, n is the callsite where the s's procedure is invoked. To preserve context-sensitivity, we select n only if the query is originally from n before entering the current procedure. If instead, s is a call statement, n is the exit (in a backward analysis) of the callee. At line 27, we perform a linear scan for the call to determine if the query can be updated in that call. We only propagate the query in the procedure if the update is possible.

Our techniques for excluding infeasible paths have been shown in Figure 13. In Figure 14, we explain how the results of infeasible path detection are reused in our fault detection. Our infeasible path detection also applies a demand-driven, query based algorithm [Bodik et al. 1997a]. Under *Branch Analysis*, we show a set of queries used to detect infeasible paths. These queries are cached at the nodes after infeasible path detection is finished. To determine the buffer safety at node 5, we propagate the constraint [Size(p)≥Len(t)]. At each node where the query arrives, we check whether the cached path conditions can impact the determination of the buffer overflow query. At node 3, we identify the impact. We therefore integrate condition [Len(t)<10] to the buffer overflow query. The updated query is resolved at node 1 as safe. This approach is more efficient than exhaustive analysis for two reasons: 1) the path conditions that are not relevant to determine the query are never collected; 2) the path conditions are symbolically simplified as a byproduct of infeasible path detection, and reused in determining different types of faults.

The loops are processed at line 28 in Algorithm 1. We classify loops into three types, based on the update of the query in the loop. We propagate the query into the loop through one iteration to determine the loop type. If the loop has no impact on the query, the query advances out of the loop. If the iteration count of the loop and the update of the query in the loop can be symbolically identified, we update the query by adding the loop's effect on the original query. Otherwise, we precisely track the loop effect on the query for a limited number of iterations (based on the user's request). Often if a loop contains multiple paths that can update a query differently, we are not able to enumerate all the potential impact of the loop on a query. We introduce a don't-know tag, "loop update unknown", to represent the undetermined loop effects on queries.

5.4.2. Mapping to Integers. We choose to map program variables to integer constraints for fault detection rather than model each bit of the variables. Our empirical experience shows that compared to bit-level precision [Xie and Aiken 2007; Babic and Hu 2008], our approach is less expensive while also is able to achieve comparable precision. By mapping to the integer domain, we not only can apply algebra and integer inequality rules to simplify the constraints, but also be able to use the integer constraint solver to help determine the satisfaction of the constraints.

The challenge of mapping program constructs to integers is to handle pointers and structures. We perform a flow-sensitive, field-sensitive, intra-procedural pointer analysis provided by the Microsoft Phoenix Compiler [Phoenix 2004]. Phoenix resolves pointers into tags that can distinguish different memory locations. Pointer analysis is performed after an ICFG of the program is built, and the results are cached for access by the later analysis. Marple conducts symbolic executions based on the cached tags. To consume aliasing relationships provided by Phoenix, Marple mainly uses two configurations: 1) consider only must aliasing and unsoundly ignore the may aliases, and 2) consider both must and some may aliasing.

To deal with non-scalar variables, such as arrays and structures of a C program or objects in C++ programs, we apply two general strategies: 1) we distinguish members of structures, e.g., using `A::i` to represent i^{th} element in array `A`, and `B::c` for the member `c` in structure `B`; and 2) we provide *attributes* to describe abstractions of a structure, e.g., Len(str) for the length of a string, and Size(list) for the size of a list. The modeling of non-scalar is performed at the statements before we construct or update a query.

5.4.3. Symbolic Substitution. Symbolic substitution is performed at the statement of interest, when the query arrives at the statement and is determined to be relevant to it (a step taken at line 11 in Algorithm 1). We determine the relevance by examining whether the variables that a query is currently tracking are defined at the current statement, either directly by a write to the destination operand, or indirectly, e.g., via side effect of a statement. Marple implements a default symbolic substitution engine, which handles basic arithmetic on integer variables. Meanwhile, Marple also provides an interface for users to supply additional rules to resolve special statements, such as library calls. When a query arrives at a statement and the current statement is determined to be relevant, Marple first searches for external rules, if no relevant rules are found, Marple applies the default integer symbolic substitution. If the statement is not an integer arithmetic in this case, we report a don't-know.

Dependent on the direction of the demand-driven analysis, Marple supports both forward and backward symbolic substitution and will apply accordingly for detecting different types of faults.

5.4.4. Don't-Know Factors. Don't-knows will be reported if the following program constructs are encountered and no external rules are provided for resolving them:

(1) Library calls: The source code of library calls is often not available. We model a set of library calls that are frequently encountered and identify others that might impact the determination of a path as don't-know.
(2) Loops and recursive calls: The iteration count of a loop or recursive call cannot always be determined statically. If we cannot symbolically reason the impact of the loop on the update of the queries, we report a don't-know.
(3) Non-linear operations: The capacity of a static analyzer is highly dependent on the constraint solver. Non-linear operations, such as bit operations, result in non-linear constraints which cannot be handled well by practical constraint solvers.
(4) Complex pointers and aliasing: Pointer arithmetic or several levels of pointer indirection challenges the static analyzer to precisely reason about memory, especially heap operations. Imprecision of *points-to* information also can originate from the path-insensitivity, context-insensitivity or field-insensitivity of a particular alias analysis used in the detection. Because of the imprecision, we can miss updating a query or incorrectly update a query, resulting unresolvable queries at the beginning of the procedure.

(5) Shared globals: Globals shared by multiple threads or by multiprocesses through shared memory are nondeterministic.

6. GENERATION OF STATIC ANALYSIS

This section introduces our techniques for automatically generating static detectors targeting specific faults using the specifications, presented in Section 4 and the demand-driven template in Algorithm 1. Specifically, we generate fault-specific **MatchFSignature** (line 4) and **MatchDSignature** (line 10) in Algorithm 1 from specifications, plug-in the generated code modules to the demand-driven template, and finally constitute individual static analyzers.

6.1. An Overview of Three Steps

The generation of static analysis follows three steps, 1) preprocessing the specification, 2) parsing, and 3) code generation. In the preprocessing step, the specification parser replaces the code signatures encapsulated in the symbol $ with constraints on the operands and operator of the corresponding program statement. The goal of this step is to convert specifications to contain only constraints and updates. As an example, consider the first pair of *CodeSignature* and *S_Constraint* from the buffer overflow specification in Figure 8. Here, by introducing an *Op* attribute, we replace code signature $`strcpy`$ with the constraint Op(s) = strcpy (the new specification variable *s* refers to a statement in the code). Correspondingly, the *S_Constraint* clause is transformed by replacing the specification variables *a* and *b* with the attributes $\text{Src}_1(s)$ and $\text{Src}_2(s)$, shown in Figure 15.

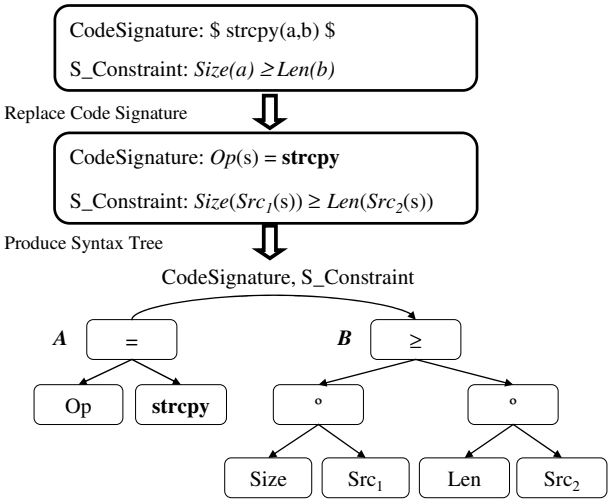


Fig. 15. Parsing: Specification to Syntax Tree

The second step aims to produce a syntax tree for each constraint or update in the specification. In the syntax tree, leaf nodes are attributes or constants, while the parents are operators for the children. As shown in Figure 15, the preprocessed buffer overflow constraint pair is converted to two syntax trees: *A* for the code signature, and *B* for the buffer overflow constraint. The symbol $\circ$ is a composition operator, which performs a function composition between Src_i and Size/Len.

The third step is code generation. The syntax tree is traversed in bottom-up order. At the leaf nodes, we find the predefined library in the Marple framework that implements the attributes. At the parent nodes, we compose the code from their children based on the semantics of the operators. Through the tree traversal, the code produced at the root implements the semantics of the tree. Figure 16 displays the process of generating code from the syntax tree produced in Figure 15. The *Step 1* box displays the implementation for the attribute *Op*. The function returns the opcode for a

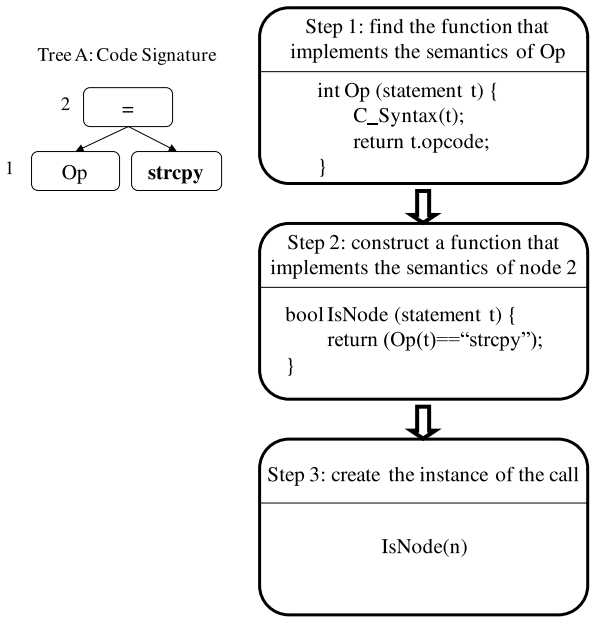


Fig. 16. Generating Analysis: Syntax Tree to Code

statement from the program. The code in *Step 2* implements the semantics of a comparison operator, $=$, which checks whether the value returned from the left leaf node equals to the one from the right node. In the *Step 3*, the call instance is produced. We further integrate the code from each syntax tree based on their relations. For example, the code generated in Figure 16 should be paired with the code generated for constructing *S_Constraint* in Figure 15.

6.2. The Algorithm for Generating Analysis

The detailed code generation process is provided in Algorithm 2. The algorithm takes a user-provided specification *spec*, and produces calls to **MatchFSignature** and **MatchDSignature**, as well as *R*, a repository of function definitions.

At line 2, we use the grammar (see Figure 9), $l.grammar$, to parse a specification. A set of pairs of syntax trees, $siglist$, is returned. Each pair of the syntax trees represents either an instance of fault signature or detection signature in the specification. The first tree in the pair is produced from the code signature, while the second represents the corresponding constraint or update.

The next step is to generate the code from the syntax trees, shown at lines 3–15 in Algorithm 2. At line 4, **CodeGenforTree** takes $sig.first$, the syntax tree of the code signature, and "n", a variable name, and generates a call that implements the semantics of the tree. The return variable, $isnode$, stores the instance of the call (function name with actual parameters), while its function definition is added to the code repository, R. See **CodeGenforTree** at lines 18–22 for details. At lines 19 and 20, we select attribute functions from the attribute library $l.attr$, and compose them based on the semantics of the operators from the syntax tree. The generated function is stored in R at line 21, and the call instance is created and returned at line 22.

Similar to the creation of $isnode$, the code for $raiseQ$ and $updateQ$ is generated. At lines 7–8, the instances of calls to $isnode$ and $raiseQ$ are integrated in an `If-Then` clause and added to fs_list. At lines 12–13, $isnode$ and $updateQ$ are combined and added to ds_list. fs_list consists of cases where a code signature of a fault is matched, and a query is raised, while ds_list consists of cases where a code signature for updating the query is matched, and the query is updated. Using the two lists, **GenSignature** produces $MatchFSignature$ at line 16 and $MatchDSignature$ at

ALGORITHM 2: Generating Analysis

```
   Input  : Specification of Fault, spec
   Output: Calls to MatchFSignature, MatchDSignature;
           A repository of function definitions, R
 1 set fs_list, ds_list to { }; initialize R = " "
 2 siglist = Parse(l.grammar, spec)
 3 foreach sig ∈ siglist do
 4 |   isnode = CodeGenforTree (sig.first, "n")
 5 |   if IsFSginature (sig) then
 6 |   |   raiseQ =CodeGenforTree (sig.second, "n")
 7 |   |   case = "If isnode then q=raiseQ;"
 8 |   |   add case to fs_list
 9 |   end
10 |   else if IsDSignature (sig) then
11 |   |   updateQ =CodeGenforTree (sig.second, "n", "q")
12 |   |   case = "If isnode then updateQ;"
13 |   |   add case to ds_list
14 |   end
15 end
16 MatchFSignature = GenSignature (fs_list)
17 MatchDSignature = GenSignature (ds_list)
18 Procedure CodeGenforTree (tree t, arglist p1, p2...)
19 alist = SelectAttrImp (t, l.attr)
20 ftree= ComposeFunc (alist, t, l.semantics)
21 Append (R, ftree)
22 return CreateCallInstance (ftree, p1, p2,...)
23 Procedure GenSignature (codelist list)
24 foreach case ∈ list do Append(case,code)
25 return code
```

line 17. The two can be plugged directly into the Demand-Driven Template at lines 4 and 10 in Algorithm 1.

The direction (backward or forward) of the analysis is chosen based on a keyword in the specification. *S_Constraint* represents a safety constraint, and thus a set of backward analysis propagation rules will be chosen to instantiate **Next** and **PropagateQ** at line 15 in Algorithm 1. On the other hand, *L_Constraint* specifies a liveness constraint, which leads to a forward analysis.

7. EXPERIMENTAL EVALUATION

We implemented Marple using YACC as well as the Microsoft Phoenix [Phoenix 2004] and Disolver [Hamadi 2002]. Our goal is to experimentally demonstrate that Marple can automatically produce path-sensitive fault detectors for multiple types of faults, and the scalability and precision of the generated detectors are comparable to manually constructed tools and those that only analyze for a specific type of fault.

The framework is implemented as a plugin to the Phoenix compiler and works on the intermediate code produced by the Phoenix front end. Thus languages that can be compiled by Phoenix, including C, C++ and C#, can be handled by our analyses. Phoenix provides pointer analysis, ICFG construction, and evaluations for integer constraints. For complex integer constraints that are not able to be handled by Phoenix and our algebra/inequality rules, we further apply Disolver to resolve them.

In our experiments, we generated an analysis that detects all of the four types of faults, namely buffer overflows, integer truncation and signedness errors, null-pointer dereferences and memory leaks. The analysis first performs an infeasible path detection and then detects each type of fault

one at a time. The detection for the first three types of faults applies a backward analysis, while the analysis for memory leak is forward.

We use a benchmark suite, consisting of 9 C/C++ programs: the first five are selected from BugBench [Lu et al. 2005] and the Buffer Overflow Benchmark [Zitser et al. 2004], and the rest are deployed mature applications. The benchmarks are chosen either 1) because they contain known faults of one of the four types for estimating false negatives of our analysis (these faults are either reported by users or discovered by runtime detectors, manual inspection or other static analyses); or 2) because they are deployed large applications for evaluating the practicality of our tool. We also use SPEC CPUINT 2000 to compare our analysis with other static detectors.

7.1. Detecting Multiple Types of Faults

In the first experiment, we run the generated analysis on the 9 benchmark programs. We evaluate the effectiveness of the analysis using four metrics: detection capability, false negatives, false positives and the path information provided for diagnosis.

Table II. Detecting Multiple Types of Faults

Benchmarks	Size (kloc)	Buffer				Integer				Pointer				Leak			
		d	mf	fp	p	d	mf	fp	p	d	mf	fp	p	d	mf	fp	p
wuftp1	0.2	4	0	0	1–11	0	-	0	-	0	-	0	-	0	-	0	-
sendmail6	0.2	0	0	1	-	3	0	3	2–2	0	-	0	-	0	-	0	-
sendmail2	0.9	4	0	0	1–4	0	-	0	-	2	-	0	1–3	1	-	0	4–4
polymorph-0.4.0	0.9	8	0	0	1–4	0	-	2	-	0	-	0	-	0	-	0	-
gzip-1.2.4	5.1	10	0	2	1–35	15	-	0	1–16	0	-	0	-	0	-	0	-
tightvnc-1.2.2	45.4	0	-	0	-	11	0	0	1–3	0	-	0	-	0	-	0	-
ffmpeg-0.4.9pre	48.1	0	-	0	-	6	0	2	1–1	1	2	0	3–3	1	-	0	2–2
putty-0.56	60.1	7	-	2	1–15	4	1	1	1–29	0	-	1	-	0	-	2	-
apache-2.2.4	268.9	0	-	0	-	2	-	2	1–2	5	-	0	1–3	0	-	0	-

In Table II, for each type of fault, we report the number of confirmed faults in Column *d*, the number of the faults that are missed in Column *mf* and the number of false positives in Column *fp*. For each program, we also give the length of the paths for the identified faults, in terms of the minimal and maximum number of procedures, shown in Column *p*. Faults here are counted using the number of statements where the constraint violations are found along some paths. We manually confirmed the data in the table.

Under *Buffer*, we show a total of 33 buffer overflows with 5 false positives. We do not miss any known buffer overflow. Among the 33 identified, 26 are newly discovered buffer overflows. The 5 false positives are all diagnosed as being on infeasible paths. As identification of infeasible paths is undecidable, we cannot exclude all infeasible paths statically. The four programs, *wuftp1*, *sendmail2*, *polymorph* and *gzip*, have also been used before to evaluate our manually constructed buffer overflow detector [Le and Soffa 2008]. The results show that the generated detector is able to report all buffer overflows detected by our manually constructed analysis. Under *Integer*, we report a total of 41 detected integer faults, 33 of which were not previously reported. We missed a fault for *putty* because we do not model function pointers. Besides insufficient infeasible path detection, imprecise pointer information is also a cause for false positives, which leads to 1 false positive for *apache* and 2 for *ffmpeg*. We identified a total of 8 null-pointer dereferences. The five identified from *apache* are cases where the pointer returned from a `malloc` function is never checked for NULL before use, which is inconsistent with the majority of memory allocations called in the program. We missed two null-pointer dereferences in *ffmpeg* as they are related to interactions of integer faults, which we did not model in this experiment. We also identified 2 memory leaks, 1 from *sendmail2* and the other from *ffmpeg*, where one cleanup procedure missed a member. For the control-centric faults of null-pointer dereference and memory leak, we only report a total of 3 false positives, compared to 15 generated by the data-centric faults. Our inspection shows that the infeasible paths related to the control-centric faults are often simple, e.g., $p \neq NULL$, and thus easily detected by our

analysis; however, integer faults and buffer overflows are more likely located along an infeasible path that is complex and not able to be identified.

Summarizing the fault detection results from the table, we identified a total of 84 faults of the four types from 9 benchmarks; 68 are new faults that were not previously reported. Inspecting these new faults, we find that many of them are located along the same paths. The dynamic approaches would halt on the first fault and never find the rest. We missed 3 known faults and reported a total of 18 false positives for the detection, mainly due to the precision of pointer analysis and infeasible path detection. The results for buffer overflow detection shows that the capability of generated detectors are comparable with manually constructed ones.

Path information about identified faults is also reported. The results under p in the table show that although the complete faulty paths can be very long, many faults, independent on the types, can be determined by only visiting 1–4 procedures. The data from *gzip* and *putty* imply that although in general, the faults were discovered by only propagating through several procedures, the longest path segments reported in the experiments contain 35 procedures, which indicate that we are able to identify faults deeply embedded in the program. Without path information, it is very difficult for manual inspection to understand how such a fault is produced.

7.2. Scalability on a General Framework

To evaluate the scalability of our technique, we collect experimental data about time and space used for analysis. The machine we used to run experiments contains 8 Intel Xeon E5345 4-core processors, and 16 GB of RAM.

Table III. Scalability

Benchmark	icfg ptr,inf	Buffer		Integer		Pointer		Leak	
		q	t	q	t	q	t	q	t
wuftp1	10.1 s	13	71.4 s	0	0	12	1.1 s	0	0
sendmail6	12.3 m	1	28.8 m	6	46.6 m	12	17.3 s	0	0
sendmail2	5.1 s	32	4.7 s	7	1.2 s	44	4.3 s	2	3.2 s
polymorph-0.4.0	1.8 m	15	8.1 s	3	6.4 s	9	1.2 s	0	0
gzip-1.2.4	25.1 m	39	18.5 m	82	70.9 s	116	6.2 s	2	7.3 s
tightvnc-1.2.2	21.9 m	21	54.9 m	1480	18.3 m	847	1.6 m	27	3.4 m
ffmpeg-0.4.9pre	49.8 m	307	88.1 m	410	33.6 m	1970	4.2 m	76	12.1 m
putty-0.56	26.4 m	150	37.9 m	79	44.1 m	256	3.2 m	14	2.4 m
apache-2.2.4	102.8 m	518	53.0 m	423	160.6 m	2730	9.6 m	21	8.2 m

In Table III, we first give the time used for preparing the fault detection, including building ICFG, and performing pointer analysis and infeasible path detection. We then list for each type of fault, the number of queries we raised, in Column q, and the time used for resolving them in Column t. The experiments show that all the benchmarks are able to finish within a reasonable time. The maximum time of 160.6 minutes is reported from analyzing *apache* for integer faults. Adding the columns under *Buffer*, *Integer*, *Pointer* and *Leak*, we obtain the total time for identifying four types of faults. For example, *apache* reports a total time of 231 minutes for fault detection, and the second slowest is *ffmpeg*, which uses 137 minutes. The time used for analysis is not always proportional to the size of the benchmark or the number of queries raised in the program. The complexity involved to resolve queries plays a major role in determining the speed of the analysis. For example, the small benchmark *sendmail6* takes a long time to finish because all the faults are related to nested loops. Another observation is that the identification of control-centric faults is much faster than the detection for data-centric faults, as for the control-centric faults we no longer need to traverse paths of loops to model complex symbolic update for the query. All of our experiments are able to finish using memory under 16 GB.

Table IV. Comparison on Memory Leak Detection

CINT2000	size (kloc)	Marple			MO_06		SC_07		Dyna-mic
		d	fp	leak_time	d	fp	d	fp	
181.mcf	1.3	0	0	2.8 s	0	0	0	0	0
256.bzip2	2.9	1	0	24.5 s	1	0	0	0	10
197.parser	6.4	0	0	26.5 s	0	0	0	0	2
175.vpr	9.6	2	0	268.5 s	0	0	0	1	47
164.gzip	10.0	2	0	8.0 s	1	2	0	0	4
186.crafty	11.3	0	0	56.3 m	0	0	0	0	37
300.twolf	15.1	17	0	25.7 m	0	0	2	0	1403
252.eon	19.3	1	0	58.2 s	-	-	-	-	380
254.gap	31.2	1	0	121.0 s	0	1	0	0	2
255.vortex	44.7	1	1	71.0 s	0	26	0	0	15
253.perlbmk	64.5	1	3	17.1 m	1	0	1	3	3481
176.gcc	128.4	27	2	7.2 h	-	-	35	2	1121

7.3. Comparison with Manually Constructed Memory Leak Detectors

In the next experiment, we compare our memory leak detection with three other tools using SPEC CPUINT 2000. The first two tools [Orlovich and Rugina 2006; Cherem et al. 2007] are static, and we name them MO_06 and SC_07. MO_06 applies a backward, exhaustive analysis. It first assumes that no leak occurs at the current program point. A memory leak is discovered if the collected information contradicts the assumption. SC_07 converts the detection for memory leak to a reachability problem using a guarded value flow graph. Neither of the tools is path-sensitive; however the impact of the conditional branch is considered in SC_07. The third tool is dynamic and we use it to compare false negatives among the static detectors, as memory leaks found in this dynamic tool are always real faults [Clause and Orso 2010].

The results are shown in Table IV. In the first and second columns, we list 12 programs and their sizes. In Column *d*, we report the number of memory leaks that are confirmed to be real, and under *fp*, we give the number of false positives. The numbers in these two columns count the memory allocation sites where a leak can occur along some paths. In dynamic analysis, however, the memory leak is reported as the number of traces that manifest the bug (see Column *Dynamic*). The numbers in this column indicate whether a memory leak exists in the programs, but it cannot be compared with the number reported under *d*.

Our experimental data show that we are able to report more memory leaks than the other static tools. We identify a total of 53 memory leaks, compared to a total of 3 shown under MO_06, and 38 under SC_07. We are able to report leaks that neither of the other tools is able to identify. For example, for *vortex*, the result from the dynamic tool shows that there exist leaks in the program; however, neither of the other two static tools reports any faults, while our analysis does. Also, we handle the C++ benchmark *eon* and report a memory leak, while the other two tools are only able to analyze C programs. We report a total of 6 false positives, shown under *fp*, compared to 29 reported by MO_06 and 6 by SC_07. We are more precise than MO_06 because we apply a path-sensitive analysis but they do not. For SC_07, our intuition is that besides using the guards on the value flow graph to help precision, other techniques are also introduced to suppress the false positives, which adversely impact the detection capability of the tool. Therefore, we are able to report more faults.

We also list the time used to detect the memory leaks. *gcc* takes the longest time, using 7.2 hours, which was still able to finish in a nightly run. Compared to the larger benchmark *apache*, *gcc* is much slower because we find many global pointers in *gcc*; also we encounter more don't-know factors when analyzing *apache* and thus is able to terminate the analysis early.

7.4. Comparison with Saturn

We also compare the analysis produced by Marple with Saturn 1.2 [Xie and Aiken 2007] on detecting null-pointer dereferences. As Marple runs on Windows while Saturn runs on UNIX, we also use SPEC CPUINT 2000, platform-independent benchmarks, for comparison. The machine we used to

run Saturn has 8 Intel E5462 4-core, 32 GB RAM, twice the memory as the machine we used to run our analysis. The two sets of experimental results are displayed in Table V under *Marple* and *Saturn*. Column *d* presents the number of true faults that are confirmed. Column *fp* displays the number of false positives. Column *ptr_time* reports the time used by our analysis to detect null-pointer dereferences after infeasible path detection is done, and *time* lists the performance of Saturn.

Table V. Comparison on Null-Pointer Dereference Detection

CINT2000	size (kloc)	Marple			Saturn		
		d	fp	ptr_time	d	fp	time
181.mcf	1.3	0	0	3.4 s	0	0	2.5 m
256.bzip2	2.9	0	0	34.3 s	-	-	EOM@1.7 m
197.parser	6.4	0	0	2.9 m	-	-	EOM@2.5 h
175.vpr	9.6	0	0	11.1 m	-	-	EOM@6.0 m
164.gzip	10.0	0	0	9.6 s	0	0	2.7 m
186.crafty	11.3	9	0	28.8 m	4	0	11.3 m
300.twolf	15.1	0	2	15.5 m	0	11	33.5 m
252.eon	19.3	0	1	3.2 m	-	-	NO C++
254.gap	31.2	0	0	38.7 m	0	9	ESF@1.3 h
255.vortex	44.7	-	-	EOM	3	14	ESF@55.5 m
253.perlbmk	64.5	-	-	EOM	0	10/45	29.7 m
176.gcc	128.4	-	-	EOM	-	-	EOM@40.5 m

Table V shows that our analysis terminates on 9 out of 12 benchmarks, and it runs out of memory on the three largest benchmarks, indicated by *EOM* under *ptr_time*. Among the 9 programs that finish, we report a total of 12 warnings, and 9 of them are confirmed as null-pointer dereferences. The root causes for the 9 faults are the same in that the program dereferences a pointer returned from a `malloc` function without verifying whether the allocation succeeds. The 3 false positives are caused by imprecision in pointer analysis and infeasible path detection.

Saturn finishes the analysis on only 5 programs. It reports out of memory on 4 programs, and segmentation faults on 2 programs (indicated as *ESF* under *time*). Saturn is not able to handle *eon* because it is written in C++. Saturn reports a total of 86 warnings, and among the 51 we inspected (we selected the first 10 warnings for *perlbmk*), 7 are confirmed as null-pointer dereferences. Our inspection shows that Saturn faces the same challenges of suppressing false positives caused by pointer analysis and infeasible detection. Besides, Saturn reports false positives caused by lack of interprocedural precision. For example, the faults we detected in *crafty* involve several procedures, which are not reported by *Saturn*. In addition, Saturn applies consistency rules to determine faults. As an example, it reports a fault if a pointer dereference followed by a check on whether the pointer is NULL. However, due to the imprecision in pointer analysis, the pointers compared in two places are not always the same, leading to false positives.

We also compare the scalability of the two fault detectors in terms of the time used to detect faults as well as the space overhead. The results show that except for *crafty* and *perlbmk*, our analysis runs faster than Saturn. We find that *crafty* and *perlbmk* contain global pointers which are accessed by many procedures; precisely tracking their interprocedural behavior is expensive. In fact, we detect null-pointer dereferences related to global points in *crafty*, while *Saturn* did not. Besides applying demand-driven analysis, we have also seen the benefit of reusing results from infeasible path detection, as during fault detection, we no longer need to resolve constraints regarding conditional branches unless they are relevant to the faults. Also, *Saturn* consumes more memory than our analysis because it models program variables in bit precision, while we propagate integer constraints. Comparing the performance displayed in Table 3, we find that in our analysis, detecting null-pointer dereference requires more memory overhead than detecting other types of faults, because the number of program points needed to be checked for detecting null-pointer dereferences are much more than the ones for identifying memory leaks.

7.5. Summary

Our evaluation demonstrates that Marple is able to detect known and new faults with reasonable false positives for real-world software. The results show that our interprocedural, path-sensitive analysis is more effective in detecting faults than path-insensitive analysis (Splint in Table 2.3.1 and the two memory leak detectors in Table IV) and region based path-sensitive analysis (Saturn in Table V). Also, our demand-driven analysis is more scalable than exhaustive analysis, as only a part of the code is visited. Our experimental results show that there is a cost in achieving generality in Marple because of the removal of specialized optimizations as well as the overhead of handling specifications; however, the analysis is still able to finish in a nightly run and thus potentially be deployed in practice.

Similar to traditional static analysis, our analysis also faces the challenges caused by imprecision in pointer analysis and infeasible path detection. However, we are able to report more useful information about the imprecision, e.g., where the imprecision potentially is located and what factors causes the imprecision.

8. RELATED WORK

Due to the importance of automatically finding faults, much work has been done for statically detecting faults from program source code. In this section, we performed a comprehensive comparison between Marple and the representative static fault detectors. A unique feature of Marple is that we apply a demand-driven analysis and detect path segments of faults. We thus also introduce how the relevant techniques, such as program slicing and demand-driven algorithms, are applied in other domains.

8.1. Static Techniques for Detecting Faults

In Table VI, we show the three commonly applied static techniques in the state-of-the-art for fault detection. Type based tools detect faults by enforcing the type safety on C. Examples include Rich [Brumley et al. 2007], CQual [David and Wagner 2004] and CCured [Necula et al. 2005]. These tools are path-insensitive and applicable for identifying integer or buffer overflows. Model checking models faults as finite automata, and checks them against abstractions of the program. Model checkers, such as MOPS [Chen and Wagner 2002], Blast [Henzinger et al. 2002] and Java PathFinder [Visser et al. 2000], report a fault as traces on the abstraction of the program, and thus are path-sensitive. Dataflow analysis determines a fault by traversing the program source and collecting the relevant information. Splint [Evans 1996] and FindBugs [FindBugs 2005] are representative path-insensitive dataflow analysis tools.

Table VI. Static Techniques for Identifying Faults

	type based	model checking	dataflow analysis
path-sensitive		MOPS [Chen and Wagner 2002] Blast [Henzinger et al. 2002] Java PathFinder [Visser et al. 2000] **see Table VII**	
path-insensitive	Rich [Brumley et al. 2007] CQual [David and Wagner 2004] CCured [Necula et al. 2005]		Splint [Evans 1996] FindBugs [FindBugs 2005]

Path-sensitive tools are summarized in Table VII in chronological order. To the best of our knowledge, none of these fault detectors have shown the scalability and generality to detect both data- and control-centric faults, as done in Marple. Compared with the other six path-sensitive tools, Marple is the only one that handles integer faults. Except for ARCHER [Xie et al. 2003], which is a buffer overflow detector, all the other tools are able to detect some types of control-centric faults, such as null-pointer dereference or memory leak. 5 out of 7 tools integrate specifications in the analysis. Metal [Hallem et al. 2002] and ESP [Das et al. 2002] provide finite automata for modeling the faults.

Prefix [Bush et al. 2000] develops a way to specify library calls. Saturn [Xie and Aiken 2007] applies a logic programming language to specify summary information at the procedure call and also inference rules for customizing an analysis. No tools listed in the table automatically generate a customized analysis from specifications as Marple does.

Table VII. Path-Sensitive Dataflow Analysis for Identifying Faults

Tools	Types of Faults			Spec	Path Traversal			Precision		Error Reports
	buf	int	fa		exhaustive	path coverage	scalability	path-sensitivity	modeling	
Prefix	×		×	model lib	×	given number	truncation	heuristically merge	ad-hoc	path
Metal			×	automata	×	all	summary	intraprocedurally	dataflow	stmt
ESP			×	automata	×	all	heuristics	heuristically merge	dataflow	path
ARCHER	×			no	×	all (timeout)	summary simple solver	intraprocedurally	linear relation	stmt
Saturn			×	summary	×	all	summary compress Booleans	intraprocedurally	limited bit-accurate	stmt
Calysto			×	no	×	configurable	compact summary	interprocedurally	bit-accurate	stmt
Marple	×	×	×	assertions flow-funcs		relevant	demand-driven caching	interprocedurally	integer,some str,containers	path seg

We also compare the tools in Table VII with regard to the way paths are traversed in the analysis. See the three metrics under *Path Traversal*. Marple is different from the other tools in that we apply a demand-driven algorithm, which allows us to explore only relevant path segments, instead of exhaustively searching for all paths.

Precision has been compared on path-sensitivity achieved in the analysis as well as the modeling of program constructs. The comparison under *path-sensitivity* shows that Calysto and Marple both achieved interprocedural, path-sensitive analysis. Summary based approaches, such as Metal, Saturn and ARCHER, do not consider interprocedural path information, and are less precise. ESP applies a heuristic to select the information that is relevant to the faults, while driven by demand, our analysis is able to determine the usefulness of the information based on the actual dependencies of variables, achieving more precision. We model integer computation and some operations of strings and C/C++ containers. Compared to Saturn and Calysto, we do not achieve bit-accurate modeling and cannot handle integer bit operations; however, the trade-off is a faster analysis.

To report an error, Prefix, ESP and Marple give path information for diagnosis, and Marple provides the path segments that are relevant to a fault. Although the fault detection is path-sensitive, other tools only report a statement where a fault occurs.

8.2. Program Slicing and Other Demand-Driven Applications

Program slicing has been applied to determine if a safety violation can be reached [Snelting et al. 2006]. The approach is to perform constraint solving on path conditions collected on program dependency graphs and determine whether an illegitimate information flow between two program points is potentially feasible. Compared to this technique, our approach is more fine grained and thus can detect more types of faults and achieve a higher precision.

Our analysis benefits from program slicing [Weiser 1981] in that we only look for program statements that the demands (i.e., queries used to determine faults) are dependent on. However, we potentially visit less nodes than slicing because our goal is to resolve the constraints in the query, i.e., the relationships of variables, which are often determined without visiting all dependent nodes. We also take two further steps beyond slicing: 1) we track the dependent nodes in a path-sensitive way, and 2) along each path, we perform a symbolic evaluation to resolve desired constraints.

Demand-driven techniques have been shown to be scalable in various domains such as pointer analysis [Heintze and Tardieu 2001], dataflow analysis [Duesterwald et al. 1997], infeasible path detection [Bodik et al. 1997a], bug detection [Le and Soffa 2008; Livshits and Lam 2003] and post-modern analysis [Manevich et al. 2004]. The demand-driven analysis can be path-sensitive [Le and Soffa 2008; Manevich et al. 2004] or path-insensitive [Duesterwald et al. 1997], forward [Livshits and Lam 2003] or backward [Bodik et al. 1997a; Manevich et al. 2004]. Our work is the first that

develops a comprehensive demand-driven framework for path-sensitive fault detection and demonstrates its effectiveness for detecting multiple types of software faults.

9. LIMITATIONS

Marple aims to practically detect faults in real-world software. The analysis is neither sound nor complete. Although we isolate don't-know factors in our part of the analysis, there is still imprecision that we are not able to track in the general fault detection. The reason is that we use the Phoenix infrastructure to model control flow and handle pointer analysis, but their techniques are not published. The sources of potential unknown imprecision caused by Phoenix are summarized as follows:

— In the version of Phoenix we used, function pointers and virtual functions are not modeled. A direct consequence is that certain part of the code may not be included in the ICFG of a program and thus not analyzed. If the query encounters such a function, the analysis terminates and reports a don't-know. In our experiments, we resolve some of the function pointers and virtual functions manually using the domain knowledge of the programs.
— We do not know the capability of Phoenix in resolving aggregate data structures. We may miss queries which should be constructed at these data types. Similarly, the query could miss an update because we do not know the variable is a structure member, where the symbolic substitution should be performed. Such imprecision can lead to both false positives and false negatives.
— Phoenix reports the may aliasing relationships without notifying why. Our manual inspection shows that in some of the cases, Phoenix returns all the heap pointers as may-aliased pointers while in other cases, when one pointer is aliasing to the other along one path but not the other, the may-aliasing is reported at the branch point. In Marple, we tried several configurations to deal with the may-aliasing: 1) only consider must-aliasing but not may-aliasing; 2) heuristically distinguish the types of may-aliasing; and 3) treat all the may-aliasing as must aliasing. We experimentally explored the three options and found that the third approach is not scalable, and the first two can bring in unknown imprecision. We used the second configuration to produce our experimental results, and the heuristic is that when Phoenix returns a large aliasing set, e.g., all of the heap pointers, we ignore the may aliasing.

In the part of the analysis we implemented, the unsoundness manifests in loss of path-sensitivity caused by two optimizations. The first optimization is *hop*, where we directly advance the query to the definition of a relevant variable in the procedure, without performing a path-sensitive propagation on the code between the entry (forward analysis) or exit (backward analysis) of the procedure and the definition of the variable in the procedure. We believe that distinguishing paths that have no impact on the query not only increases the analysis overhead but also provides overwhelming useless information for fault diagnosis.

Our second optimization lies in handling of branches. Figure 14 explained that we use the intermediate results cached from branch correlation analysis to determine the impact of the path conditions on the queries. In the optimization, we do not check, at each node, whether the conditional branch is relevant to the query; instead, we select nodes where the impact potentially is able to be determined. For example, in Figure 17, under *Fault Detection*, we can check the path condition at nodes 4 and then 3, but not nodes 2 and 1, and still be able to correctly determine the query resolution. We found through our empirical experience, branches near where a query is updated often can impact the query; we therefore only check the nodes that are n steps away from where the query is updated. Properly choosing n can help balance the precision and scalability of the analysis. As an example, under *Heuristic* in Figure 17, the imprecision can occur if we set to only check the path condition at the branch node. In this case, condition [`i<10`] at node 4 will not be integrated, and query [Size(p) $\geq$ Len(t)] is determined as faulty at node 1, leading to a false positive.

It should be noted that the above discussions mainly focus on the analysis. The soundness and completeness of the framework are also determined by the specifications applied. For example,

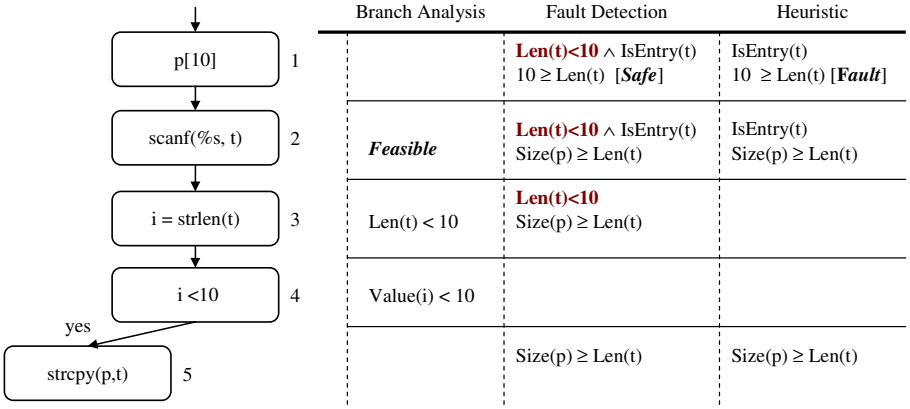


Fig. 17. Potential Imprecision for Dealing with the Path Conditions

missing a fault signature in the specification, we potentially miss bugs and an incorrect detection signature potentially causes false positives or false negatives.

10. CONCLUSIONS

This article presents Marple, a demand-driven, static framework that detects path segments of faults. The framework integrates a set of techniques to address the challenges of precision, scalability and generality. The key for precision is to map the definition of a fault to integer constraints and perform an interprocedural, context-sensitive, path-sensitive symbolic analysis to resolve the integer constraints. The scalability of the path-sensitive analysis is achieved via a demand-driven algorithm that focuses on only the fault-relevant code. To make the precise and scalable analysis generally applicable, we develop a specification technique and an algorithm to automatically produce static analyses that target user-specified faults. Our experimental evaluation demonstrates the competitive detection capability, scalability and precision of Marple in detecting buffer overflows, integer faults, null-pointer dereferences and memory leaks for a set of real-world applications. Although in this article we mainly focus on traditional faults, with our technique, users can write specifications and identify their own defined faults. Our future work includes further investigation of the interactions of different types of faults and also exploration of the potential parallelism that exists for computing query resolutions.

REFERENCES

BABIC, D. AND HU, A. J. 2008. Calysto: scalable and precise extended static checking. In *ICSE'08: Proceedings of the 30th international conference on Software engineering*.

BODIK, R., GUPTA, R., AND SOFFA, M. L. 1997a. Interprocedural conditional branch elimination. In *PLDI'97: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

BODIK, R., GUPTA, R., AND SOFFA, M. L. 1997b. Refining data flow information using infeasible paths. In *FSE'05: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

BRUMLEY, D., CKER CHIUEH, T., JOHNSON, R., LIN, H., AND SONG, D. 2007. RICH: Automatically protecting against integer-based vulnerabilities. In *NDSS'07: Proceedings of the 14th Symposium on Network and Distributed Systems Security*.

BUSH, W. R., PINCUS, J. D., AND SIELAFF, D. J. 2000. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*.

CHEN, H. AND SHAPIRO, J. S. 2004. Using build-integrated static checking to preserve correctness invariants. In *Proceedings of the 11th ACM conference on Computer and communications security*. CCS '04.

CHEN, H. AND WAGNER, D. 2002. MOPS: an infrastructure for examining security properties of software. In *CCS'02: Proceedings of the 9th ACM Conference on Computer and Communications Security*.

CHEREM, S., PRINCEHOUSE, L., AND RUGINA, R. 2007. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*.

CLAUSE, J. AND ORSO, A. 2010. Leakpoint: pinpointing the causes of memory leaks. In *ICSE'10: Proceedings of the 32nd International Conference on Software Engineering*.

DAS, M., LERNER, S., AND SEIGLE, M. 2002. ESP: path-sensitive program verification in polynomial time. In *PLDI'02:Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*.

DAVID, R. J. AND WAGNER, D. 2004. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th conference on USENIX Security Symposium*.

DILLIG, I., DILLIG, T., AIKEN, A., AND SAGIV, M. 2011. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. PLDI '11. ACM, New York, NY, USA, 567–577.

DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. 1997. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*.

DWYER, M. B., ELBAUM, S., PERSON, S., AND PURANDARE, R. 2007. Parallel randomized state-space search. In *ICSE'07: Proceedings of the 29th international conference on Software Engineering*.

EVANS, D. 1996. Static detection of dynamic memory errors. In *PLDI'96: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*.

FINDBUGS. 2005. http://findbugs.sourceforge.net/.

HACKETT, B., DAS, M., WANG, D., AND YANG, Z. 2006. Modular checking for buffer overflows in the large. In *ICSE'06: Proceeding of the 28th International Conference on Software Engineering*.

HALLEM, S., CHELF, B., XIE, Y., AND ENGLER, D. 2002. A system and language for building system-specific, static analyses. In *PLDI'02, Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*.

HAMADI, Y. 2002. Disolver : A Distributed Constraint Solver. Tech. Rep. MSR-TR-2003-91, Microsoft Research.

HEINTZE, N. AND TARDIEU, O. 2001. Demand-driven pointer analysis. In *PLDI'01: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*.

HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy abstraction. In *POPL'02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

LE, W. AND SOFFA, M. L. 2008. Marple: a demand-driven path-sensitive buffer overflow detector. In *FSE'08: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*.

LE, W. AND SOFFA, M. L. 2010. Path-based fault correlation. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of software engineering*.

LIVSHITS, V. B. AND LAM, M. S. 2003. Tracking pointers with path and context sensitivity for bug detection in c programs. In *FSE'03: Proceedings of 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

LU, S., LI, Z., QIN, F., TAN, L., ZHOU, P., AND ZHOU, Y. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Proceedings of Workshop on the Evaluation of Software Defect Detection Tools*.

MANEVICH, R., SRIDHARAN, M., ADAMS, S., DAS, M., AND YANG, Z. 2004. PSE: explaining program failures via postmortem static analysis. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*.

MICROSOFT PREFAST. Keynote talk. In *"http://www.microsoft.com/whdc/devtools/tools/prefast.mspx"*.

NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2005. CCured: type-safe retrofitting of legacy code. *ACM Transactions on Programming Languages and Systems Volume 27 Issue 3*.

ORLOVICH, M. AND RUGINA, R. 2006. Memory leak analysis by contradiction. In *SAS'06: Proceedings of the 13th International Static Analysis Symposium*.

PHOENIX. 2004. http://research.microsoft.com/phoenix/.

POLYSPACE. 2001. http://www.mathworks.com/products/polyspace/.

SNELTING, G., ROBSCHINK, T., AND KRINKE, J. 2006. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol. 15*, 410–457.

STROM, R. E. AND YEMINI, S. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transaction Software Engineering 12,* 1, 157–171.

VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S. 2000. Model checking programs. In *ASE'00: Proceedings of the 15th IEEE international conference on Automated software engineering*. 3.

WEISER, M. 1981. Program slicing. In *ICSE 81: Proceedings of the 5th international conference on Software engineering*.

XIE, Y. AND AIKEN, A. 2007. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transaction Program Language System 29,* 3.

XIE, Y., CHOU, A., AND ENGLER, D. 2003. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

ZITSER, M., LIPPMANN, R., AND LEEK, T. 2004. Testing static analysis tools using exploitable buffer overflows from open source code. In *FSE'04: Proceedings of the 12th International Symposium on Foundations of Software Engineering*.

A. SPECIFICATIONS FOR INTEGER FAULTS AND NULL-POINTER DEREFERENCES

A.1. Integer Fault

Figure 18 provides the specification for detecting integer faults. The fault signature consists of five pairs, among which, the first three are for detecting integer overflows/underflows, the fourth for integer truncation problems, and the last one is for finding integer signedness conversion violations. For integer overflows/underflows, the safety constraints require that the result of integer arithmetics should fall in the range that the destination integer can store, specified as [TMin(x),TMax(x)]. Attribute TMin(x) represents the minimum value the integer x possibly holds under the type, and TMax(x) is the maximum. As TMin(x) and TMax(x) are known when the type of x is given, to resolve the constraint, the analysis needs to identify the value or range for the source operands of y and z. Similarly, the fourth definition says that when integer y is assigned to integer x, and y has a larger integer width (more bits used in the machine to represent the integer) than x, we need to check if the value of y can be hold by the destination operand. Finally, the last rule specifies that for every use of integer x, if its definition, Def(x), has a different signedness bit, a signedness conversion occurs; we therefore need to check before the conversion whether the value of the x satisfies the range constraint.

To detect integer faults, we can reuse the symbolic execution rules developed for buffer overflow to resolve integer constraints, as the two both require the tracking of the value or range of integers. Under *Detection*, we list a set of frequent used rules for determining both integer and buffer safety violations. It should be noted that we have integrated basic integer arithmetic symbolic rules in the general symbolic analysis engine. Here, the most of the rules are to resolve a library call or an array reference. To determine which rules to use, we will match the types of variables under tracking with the type of variables applied in specification.

A.2. Null-Pointer Dereference

Buffer overflow and integer fault are data-centric faults. Here we show specifications for control-centric faults of null-pointer dereference. In Figure 19, the attribute *MatchOperand* under *Fault* describes the cases where the pointer dereference occurs. This attribute examines operands in the statement to match the pointer dereference signatures of `*a`, `a->e`, or `a[d]`. If such program points are matched, we should examine before the pointer dereference whether the pointer is a non-NULL.

Under *Detection*, the first two pairs of *CodeSignature* and *Update* imply that when a NULL or a constant string is assigned to the pointer, the constraint is determined. The third rule specifies a symbolic substitution rule between two pointers when aliasing is detected.

Vars		*Vint* x,y,z,d; *Vbuffer* b;
Fault		
	CodeSignature	$x=y+z$
	S_Constraint	(Value(y)+Value(z)) ∈ [TMin(x),TMax(x)]
or		
	CodeSignature	$x=y-z$
	S_Constraint	(Value(y)-Value(z)) ∈ [TMin(x),TMax(x)]
or		
	CodeSignature	$x=y*z$
	S_Constraint	(Value(y)*Value(z)) ∈ [TMin(x),TMax(x)]
or		
	CodeSignature	$x=y$ && IntegerWidth(y)>IntegerWidth(x)
	S_Constraint	Value(y) ∈ [TMin(x),TMax(x)]
or		
	CodeSignature	IsUse(x) && Signedness(x)≠ Signedness(Def(x))
	S_Constraint	Value(Def(x)) ∈ [TMin(x),TMax(x)]
Detection		
	CodeSignature	$strcpy(a,b)$
	Update	Len(a):= Len(b)
or		
	CodeSignature	$strcat(a,b)$
	Update	Len(a):= Len(a)+Len(b)
or		
	CodeSignature	$a[d]=e$ && Value(e)='\0'
	Update	(Len(a)>Value(d) ‖ Len(a)=undef)
		↦ Len(a):= Value(d)
or		
	CodeSignature	$d=strlen(b)$
	Update	Value(d):= Len(b)
or		
	CodeSignature	$perror(d)$
	Update	skip
or		
	CodeSignature	$scanf(e,d)$
	Update	IsEntry(d)
or		
	CodeSignature	$d=atoi(a)$
	Update	Value(d):= Len(a)
or		
	CodeSignature	$a[d]=f$
	Update	Array(a,d):= Value(f)
or		
	CodeSignature	$a[d]=f$ AND Value(f)=0
	Update	Len(a):= Value(d)
or		
	CodeSignature	$sprintf(a,$c_1$, c_2, ..., c_n)$
	Update	Len(a):= Len(c_1)+ Len(c_2)+...+Len(c_n)-2*Value(n)

Fig. 18. Detecting Integer Overflows/Underflows, Truncation and Signedness Conversion Problems

Vars		*Vptr* a,b, c; *Vint* d; *Vany* e;
Fault		
	CodeSignature	MatchOperand(`*a`, `a->e`, `a[d]`) $\neq \emptyset$
	S_Constraint	Value(a)$\neq$ NULL
Detection		
	CodeSignature	$`a=c`$&&IsConstStr(c)
	Update	$\wedge$ Value(a) $\neq$ NULL
or		
	CodeSignature	$`a=NULL`$
	Update	Value(a):= NULL
or		
	CodeSignature	$`a=b`$
	Update	Value(a):= Value(b)

Fig. 19. Detecting Null-Pointer Dereferences