Generating Analyses for Detecting Faults in Path Segments

Wei Le and Mary Lou Soffa
Department of Computer Science
University of Virginia
Charlottesville, VA, USA
{weile, soffa}@virginia.edu

ABSTRACT

Although static bug detectors are extensively applied, there is a cost in using them. One challenge is that static analysis often reports a large number of false positives but little diagnostic information. Also, individual bug detectors need to be built in response to new types of faults, and tuning a static tool for precision and scalability is time-consuming. This paper presents a novel framework that automatically generates scalable, interprocedural, path-sensitive analyses to detect user-specified faults. The framework consists of a specification technique that expresses faults and information needed for their detection, a scalable, path-sensitive algorithm, and a generator that unifies the two. The analysis produced identifies not only faults but also the path segments where the root causes of a fault are located. The generality of the framework is accomplished for both data- and control-centric faults. We implemented our framework and generated fault detectors for identifying buffer overflows, integer violations, null-pointer dereferences and memory leaks. We experimentally demonstrate that the generated analyses scales to large deployed software, and its detection capability is comparable to tools that target a specific type of fault. In our experiments, we identify a total of 146 faults of the four types. While the length of path segments for the majority of faults is 1–4 procedures, we are able to detect faults deeply embedded in the code across 35 procedures.

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

General Terms
Algorithms, Experimentation, Reliability, Security

Keywords
Demand-Driven, Path Segment, Specification, Generate Analysis

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 applying static tools in practice is the high manual cost. One reason is that static analysis typically reports high false positives, requiring manual effort to confirm the warnings. Diagnosing these warnings can be challenging as useful information, such as program paths where a fault potentially occurs, is likely missing. Also, static fault detectors are often difficult to scale for large software; as a result, manually developed annotations sometimes are inserted to help with the scalability [11, 13]. Furthermore, since tools hardcode heuristics targeting one type of fault or even a particular program, repeated effort has to be invested to build individual analyzers.

Ideally, we should develop static tools that are: 1) precise, not only reporting a reasonable number of false positives and false negatives but also identifying program paths where a fault potentially occurs to help diagnose warnings; 2) scalable, terminating analyses with reasonable code coverage; and 3) general, in that the strategies developed for precision and scalability are commonly applicable and the definitions of various faults can be easily integrated.

Previous research has addressed some of these goals. For precision, path-sensitive analysis is applied [8, 13, 26]. In path-sensitive analysis, identified infeasible paths are excluded, and program facts collected along different paths for determining faults are not merged, avoiding a loss of precision. The problem is that, in existent techniques, the scalable path-sensitive analysis is achieved via either heuristics or summary-based approaches, compromising precision and generality. For example, ESP [8] achieves a polynomial time verification on identifying paths of typestate violations. However, its scalability is accomplished via an assumption on how a programmer might program typestate transitions in practice; as a result, the heuristics applied only shown effective in detecting typestate errors. Tools such as Saturn [26] apply a summary based program analysis. That is, path-sensitivity is only achieved within a procedure, and then the intraprocedural analysis results are composed at the whole program level. This modular technique is not effective for faults whose detection requires a large amount of summary information or the tracking of interprocedural information, such as global side effects.

One challenge of developing an ideal static analyzer is that there exists an exponential number of paths in a program, and exhaustively examining all these paths to find faults, as most current tools do, is not practical. Our insight is that not every program path is relevant to faults of interest, and even for a faulty program path, not all statements along the path are attributable to the production of a fault. Therefore, by directing static analysis to the code that is likely to contain a fault and reporting path segments that are relevant to the faults, we potentially achieve both scalable and precise fault detection. Our hypothesis is also that such analysis is generally applicable, and can be modularized to easily extend to different
types of faults.

In this paper, we present a practical framework that automatically produces analyses to statically detect faults of various types. A novelty is that for both precision and scalability, our analysis focuses on program paths, in particular, the segments along the paths, which lead to the faults. For generality, we develop a specification technique and an algorithm that enables automatic generation of desired static analyses.

Specifically, precision is achieved via an interprocedural, path-sensitive, symbolic analysis. Path segments that are relevant to the production of faults are reported, providing both the focus and necessary context for a code reviewer to diagnose the faults. We prioritize warnings based on the confidence of the analysis, and report factors that potentially cause the imprecision. The framework also provides an interface for supplying external information to resolve the warnings that static analysis cannot resolve.

For scalability, our framework integrates a general, demand-driven algorithm which formulates the demand of fault detection into queries regarding whether potentially faulty statements in a program are safe. Path-based, demand-driven analysis can achieve the same precision as exhaustive analysis, but is more efficient. Guided by the demand, the analysis knows which information is useful for determining the fault. Orthogonally, we exploit the opportunities of reusing intermediate results to further speed up the analysis. Instead of evaluating path conditions along each individual path, we perform a branch correlation analysis before fault detection. The constraints regarding conditional branches are resolved once and will be reused in detecting different types of faults. Intermediate queries used for determining faults are also cached for reuse.

Generality is achieved via a framework that integrates a general demand-driven analysis, a specification language and an algorithm that enables automatic generation of individual detectors from specifications. The general analysis incorporates our design decisions for addressing precision and scalability. The specification language expresses the definition of a fault and also the information needed to detect the fault. The framework currently handles both data- and control-centric faults that have the following two features: 1) they are observable at certain types of program statements; and 2) on the paths to such observable points, we are able to identify the types of statements that contribute to the failure. By identifying such observable points, we can construct a query at those points, inquiring whether the fault can occur along any paths that lead to the points. Similarly, given contributing points, we know where to collect information to resolve the query and determine the fault. Our specification provides both observable and contributing points, and also specifies the actions an analysis should take at these program points. To generate an analysis, we translate the specification to code modules, and plug the modules into the demand-driven, symbolic analysis template to produce desired fault detectors.

We implemented our framework, Athena, using the Microsoft Phoenix infrastructure [23] and the Solver constraint solver [15]. We experimentally show that Athena is able to generate fault detectors whose detection capability is comparable with manually constructed tools that target a particular type of fault. We also demonstrate that our generated analysis can identify buffer overflows, integer faults, null-pointer dereferences and memory leaks with interprocedural path-sensitive precision, and is scalable to at least 250 k lines of code. Our results show that while we are able to discover faults deeply nested in 35 procedures, the path segments for faults are generally short, 1–4 procedures, providing focus for diagnosis.

The contributions of the paper include:

- a framework that enables automatic generation of fault detectors for multiple types of faults,

- a scalable algorithm that detects both control- and data-centric faults and returns path segments of the faults,
- a specification language that specifies both the definition and detection of the faults, and
- implementation and experimentation that demonstrate the effectiveness of our analysis in detecting buffer overflows, integer faults, null-pointer dereferences and memory leaks.

In the rest of the paper, we present an overview of the framework in Section 2. We then explain components in the framework: specification techniques in Section 3, demand-driven, path-sensitive symbolic analysis in Section 4, and the automatic generation of analyses in Section 5. Our experimental results are shown in Section 6, followed by the related work in Section 7. The conclusion is given in Section 8. In this paper, for any discussions that are related to the code, we use C/C++ as an example.

2. AN OVERVIEW OF ATHENA

Athena takes a user-provided specification, and generates interprocedural, path-sensitive analyses that identify path segments of the specified faults. In Figure 1(a), we present the five components of the Athena framework. The Specification Language consists of syntax and semantics of the language. The Parser and the Analyzer Generator translate the specification and produce the parts of the analysis that target the specified fault. A general, path-sensitive, demand-driven algorithm is developed in a Demand-Driven Template, which implements our design decisions for handling the challenges of precision and scalability. The Specification Repository consists of specifications we constructed for a set of common faults, including buffer overflows, integer faults, 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 Athena.

As shown in Figure 1(b), given a specification, the Parser first produces a set of syntax trees. Based on the semantics of the specification, the Analyzer Generator generates the code modules that implement the rules for determining the specified faults. The code modules are plugged into the Demand-Driven Template to produce the analyzer. The specifications for multiple types of faults can be integrated to generate an analysis that handles different types of faults. The advantage of such an analysis is that we can reuse the intermediate results—e.g., feasibility or aliasing information, for
analyzing different types of faults—and also explore the interactions of different types of faults [18].

3. SPECIFICATION LANGUAGE

Our specification language expresses both a fault and the information used to statically detect the fault. We first discuss how we model faults and their detection.

3.1 Modeling Faults and their Detection

To identify a fault statically, we need to know what is 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.

We define a fault signature to specify a fault, and a detection signature for expressing program points that can contribute to the fault. Based on our definition of a fault, we identify two elements to constitute a fault signature: the program point and a property that must hold at the point. We specify program points, using code signatures of particular types of statements, and properties, using constraints, which define conditions on program states. The key elements to compose constraints are attributes. Attributes represent an abstraction of program state, and specify properties of program objects, such as program variables or statements. For instance, an attribute can be a value, range, or typestate of individual program variables, or relations among multiple variables.

Here, we give two examples. 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 define code signatures of "open file", and a constraint as "a close has to follow the open".

Besides the above examples, we later show that our technique can also model traditional faults of integer violation, null-pointer dereference and memory leak. 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 or range of program variables. The types of faults Athena can handle are similar to the ones detected by traditional static analysis. The determinant factor is whether a fault can be mapped to a set of constraints and the program points where constraints should hold.

To determine the detection signature, we need to understand how a fault is produced dynamically. At runtime, a set of changes of program states at certain program points along the execution lead to the violation of the property constraints. Therefore, to statically determine the violation of constraints, we need to provide, in detection signatures, types of program points that have an impact on producing a fault, and their impacts on constraints.

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, 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. Since static analysis can be either backwards or forwards, we should follow a direction where the information that determines the resolutions of the constraints is located.

<table>
<thead>
<tr>
<th>Vars</th>
<th>FaultSignature</th>
<th>Vbuffer a, b; Vint d; Vany e;</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeSignature</td>
<td>$strcpy(a,b)$</td>
<td>S_Constraint Size(a)≥Len(b)</td>
</tr>
<tr>
<td>or</td>
<td>$memcpy(a,b,d)$</td>
<td>S_Constraint Size(a)≥min(Len(b), Value(d))</td>
</tr>
<tr>
<td>or</td>
<td>$a[d]=e$</td>
<td>S_Constraint Size(a)&gt;Value(d)</td>
</tr>
<tr>
<td>DetectionSignature</td>
<td>$strcpy(a,b)$</td>
<td>Len(a):=Len(b)</td>
</tr>
<tr>
<td>or</td>
<td>$memcpy(a,b)$</td>
<td>Len(a):=Len(a)+Len(b)</td>
</tr>
<tr>
<td>or</td>
<td>$a[d]=e$ &amp; Value(e)=&quot;'(0'</td>
<td>Len(a)&gt;Value(d)</td>
</tr>
<tr>
<td>or</td>
<td>$strlen(b)$</td>
<td>Value(d):=Len(b)</td>
</tr>
</tbody>
</table>

**Figure 2:** Partial Buffer Overflow Specification

3.2 Specification and Examples

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 the program points and the 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.

The specification language provides a set of commonly used attributes, as well as the operators computation, comparison, composition and command. The attributes take program variables, and return an integer, Boolean, or set. The semantics of attributes are predefined and implemented as library calls on the Athena 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. The language also provides facilities to pair code signatures with constraints or update rules to compose the fault or detection signature.

We show a buffer overflow specification in Figure 2. The keyword Vars defines variables used in the specification. Under FaultSignature, the keyword CodeSignature provides a set of program points where the buffer constraints have to be enforced. Three examples are the library calls 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. It can be specified using a comparator "≥" on attributes of Size(a), the size of buffer a, and Len(b), the length of the string b. The role of variables such as a and b is to locate the operands in the code signature for constructing constraints.

Under DetectionSignature, 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 the symbol \( \rightarrow \). 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,
<table>
<thead>
<tr>
<th>Vars</th>
<th>Vptr a, b; Vint c</th>
</tr>
</thead>
<tbody>
<tr>
<td>FaultSignature</td>
<td></td>
</tr>
<tr>
<td>CodeSignature</td>
<td>$a = \text{malloc}(c)$</td>
</tr>
<tr>
<td>L_Constraint</td>
<td>TypeState(a) $\Rightarrow$ 1</td>
</tr>
<tr>
<td>DetectionSignature</td>
<td></td>
</tr>
<tr>
<td>CodeSignature</td>
<td>$\text{free}(a)$</td>
</tr>
<tr>
<td>Update</td>
<td>TypeState(a) $\Rightarrow$ 1</td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
<tr>
<td>CodeSignature</td>
<td>$a = b$</td>
</tr>
<tr>
<td>Update</td>
<td>$\text{Ref}(a) \Rightarrow 1 \rightarrow \text{TypeState}(a) = 0$</td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
<tr>
<td>CodeSignature</td>
<td>IsEnd(a)</td>
</tr>
<tr>
<td>Update</td>
<td>$\text{Ref}(a) \Rightarrow 0 \rightarrow \text{TypeState}(a) = 0$</td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
<tr>
<td>CodeSignature</td>
<td>$b = a$</td>
</tr>
<tr>
<td>Update</td>
<td>$\text{Ref}(a) = \text{Ref}(a) + b$</td>
</tr>
</tbody>
</table>

Figure 3: Partial Memory Leak Specification

Len(a) = $\text{undef}$, we can assign the string length of $a$ with the value of $b$. It should be noted that Athena integrates a symbolic substitution module to automatically handle integer computation, e.g., using rules Value(x) := Value(y) for the program point $x = y$. 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.

We also present a specification for memory leak in Figure 3. The constraint for memory leak is that a memory allocation is safe only if a free of the memory is invoked in the future. It is a liveness constraint and defines a control-centric fault. In the specification, we use the attribute TypeState(a) to record the order of operations performed on the section of memory tracked by $a$. The constraint under FaultSignature says that when TypeState(a) equals 1, the leak does not occur. Under DetectionSignature, 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, or its scope ends. 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 reference set contains a set of pointers that currently point to the tracked memory). The last rule in the specification adds the aliasing pointer to the reference set.

### 3.3 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 overflow 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. Intuitively, a finite automata, FA, can potentially be converted to our specification: the fault signature can be derived from the end states and their incoming edges of FA, and the detection signature can be obtained from transitions between states in FA. To extend Athena 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 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 fault signature, and modeled about 50 different types of program statements, many of which are C libraries related to string manipulations. 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 memory leak.

### 4. DEMAND-DRIVEN TEMPLATE

The above specifications are integrated to a general, predefined, static analysis template for detecting specified faults. In the template, we develop an interprocedural, demand-driven, path-sensitive analysis for achieving the desired scalability and precision. The analysis detects faults by raising queries at the statements where faults potentially occur, inquiring whether the constraints of correctness would hold. To determine the resolutions of constraints, the analysis performs a path traversal on the code that is relevant to the fault, either backward (for safety constraints) or forward (for liveness constraints).

#### 4.1 An Example

We first use an example to intuitively explain how the analysis works to determine integer fault and null-pointer dereference. In Figure 4, 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, an integer signedness conversion occurs for integer $x$ to match the interface of malloc. We raise a query $\text{Value}(x) \geq 0$ 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 $\text{Value}(x) \geq 0$ via a symbolic substitution. Along branch (8, 7), the query encounters a constant assignment and is resolved to $1024 \geq 0$ as safe. Along the other branch (8, 6), the analysis derives the information $x \leq -1$ from the false branch, which implies the constraint $\text{Value}(x) \geq 0$ is always false. Therefore, we resolve the query as unsafe. Path segment (6, 8, 10, 11) 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. We perform an infeasible path analysis [2] before fault detection, and the identified infeasible paths are marked on the interprocedural control flow graph (ICFG), as ip1 and ip2 shown in Figure 4. To detect the null-pointer dereference, the analysis starts at a pointer dereference discovered at node 13. Query $\text{Value}(p) \neq \text{NULL}$ is constructed, meaning the pointer $p$ should be non-NULL before the dereference at node 13 for correctness. At branch (13, 12), the query encounters the end of the infeasible path and records ip1 in progress. Along one path (13, 12, 9), the propagation no longer follows the infeasible path and thus the query drops ip1. The query is resolved...
as safe at node 9, assuming memory allocation succeeds and malloc returns a non-NULL p. Along the other path (13–10), no update occurs until the end of ip2 is met at node 10. The query thus records ip2 in progress. When the query arrives at branch (5,4), the start of ip1 is discovered, showing the query traverses an infeasible path. The analysis terminates. Similarly, the propagation halts at branch (5,3) for traversal of ip2. The analysis reports node 13 as safe for null-pointer dereference.

### 4.2 Demand-Driven, Path-Sensitive Symbolic Analysis: a General Algorithm

The Demand-Driven Template, shown in Algorithm 1, is a skeleton of our demand-driven algorithm, which mainly provides query propagation rules that are generally applicable for identifying different types of faults. The skeleton has "holes", MatchFSignature at line 4 and MatchDSignature at line 10, where the fault-dependent information is missing. MatchFSignature examines whether a given program statement matches the code signature of a fault; if it does, a query is constructed using the constraints in the fault signature. MatchDSignature determines whether a given statement matches the code signature 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.

At line 1, the analysis first builds an ICFG for the program. The pointer analysis is performed to determine aliasing information and model C/C++ structures. A branch correlation analysis is also conducted to detect infeasible paths [2].

Lines 3–16 of Algorithm 1 presents the demand-driven analysis that detects faults.

The analysis first performs a linear scan of statements in the ICFG to match the fault signature. If the match succeeds, a query is returned and added to a worklist at line 6. The main component of a query is an integer constraint, inquiring whether a violation can occur. The query also includes in-progress information tracked by the analysis, such as to which nodes it has been propagated.

![Figure 4: Detecting Different Types of Faults](image_url)

After the demand is constructed, a path-sensitive analysis is performed on the code reachable from where the query is raised. At line 10, if a statement is matched to a detection signature, 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 gives details of the query evaluation. At line 19, we first simplify the constraints using the algebraic identities and inequality properties. An integer constraint solver is also called to further determine the resolution of the constraint. If the constraint returns true, the safety rule is always satisfied independent of the input; 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. The idea of don't-know is that we do not want the analysis to continue tracking potentially imprecise results. Instead, we notify users with the location and factors that cause don't-knows. The analysis terminates for the query if its resolution is determined.

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. 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 4.3).

The analysis terminates when the resolutions for all the queries in the worklist are determined. At line 17, we report path segments that are traversed by the queries. The path segments start where a query is raised and end where the resolution of the query is determined. Along a path segment, the constraints of a fault 1 are always resolved as false, which implies that as long as the execu-
tion traverses the path segment, the fault can be triggered, or 2) report violations on some user input, which says any execution that crosses the path segments with a proper input can trigger the fault.

4.3 Details for Precision and Scalability

Path-sensitive analysis can differ in precision. Our path-sensitivity has the following features: 1) we track interprocedural paths (handled in Next at line 15 and InterproceduralMove at line 27 in Algorithm 1); 2) queries are not merged at branches and the relevant conditional branches are integrated (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 at the beginning of the procedure, n is the callsite where 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 are shown in Figure 4. In Figure 5, 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 [2]. Under Column Branch Analysis, we show a set of queries used to detect infeasible paths; these queries are cached at the nodes after infeasible path detection. To determine the buffer safety at node 5, we propagate constraint \[\text{Size}(p) \geq \text{Len}(t)\]. 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 \[\text{Len}(t) \leq 10\] 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 conditions that are not relevant to determine the query are never collected; 2) the path conditions are symbolically evaluated 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. GENERATING ANALYSIS

To instantiate the demand-driven template in Algorithm 1, we provide fault-specific information via MatchFSignature (line 4) and MatchDSignature (line 10). This section presents an algorithm that automatically generates the two modules from a specification.

5.1 An Overview of the Approach

In a specification, there are three important language constructs: 1) code signatures, 2) constraints and updates, and 3) keywords that pair the two.

As a first step, the specification parser replaces the code signatures encapsulated in the symbol $ with constraints on the operands and operator of the corresponding program statement. A specification is thus converted to only contain constraints and updates. In the second step, each constraint or update is parsed into a syntax tree, whose leaf nodes are attributes or constants, while the parents are operators for the children. Finally, during code generation, the syntax tree is traversed in a bottom-up order. At the leaf nodes, we find the code that implements the predefined attributes from the attribute library in the Analyzer Generator. 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. We further integrate the code from each syntax tree based on their relations. For example, the code implements matching code signature should be paired with the code for constructing the corresponding constraint or update.

5.2 The Algorithm for Generating Analysis

Algorithm 2 provides in detail the code generation process. 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, l.grammar, to parse a specification. A set of pairs of syntax trees, siglist, is returned. Each pair of the syntax trees represents either a fault signature or a 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. As an example, consider the first pair of CodeSignature and S_Constraint from the buffer overflow specification in Figure 2. In Figure 6, the parser introduces the attribute Op(s) to represent the operator of statement s, and SrcC(_s) for the i-th operands. Specification variables a and b, which represent the locations of operands, are replaced accordingly for both the code signature and the constraint. The constraints are converted to syntax trees: A for the code signature, and B for the buffer overflow constraint. The symbol $ in the figure is a composition operator, which performs a function composition between Src and Size/Len.

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 Lattr, 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.

Figure 7 displays the process of generating code from the syntax tree produced in Figure 6. The Step 1 box displays the implementation for the attribute Op. The function returns the opcode for a statement from the program. The code in Step 2 implements the semantics of a comparison operator, $=$, which checks whether the value returns from the left leaf node equals to the one from the right node. In the Step 3, the call instance is produced.

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 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.

6. EXPERIMENTAL EVALUATION

We experimentally evaluated our framework to demonstrate that Athena can automatically generate path-sensitive fault detectors to identify 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.

6.1 Experimental Setup

Athena is implemented as a plugin to the Phoenix compiler [23]. Languages that can be compiled by Phoenix, including C, C++, and C#, can be handled by our analysis. Phoenix provides pointer analysis, ICFG construction, and a part of the attribute library. We apply Dissolver [15] to evaluate integer constraints. We use YACC to implement the specification parser. In our experiments, we generate an analysis that detects all 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 feasible path detection, and then detects each type of fault one at a time. The detection for the first three types of faults applies a feasible path detection, and then detects each type of fault one at a time. The detection for the first three types of faults applies a feasible path detection, and then detects each type of fault one at a time. The detection for the first three types of faults applies a feasible path detection, and then detects each type of fault one at a time.

The first five are selected from BugBench [20] and the Buffer Overflow Benchmark [28], 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; or 2) because they are deployed large applica-
tions for evaluating the practicality of our tool. We also use SPEC CPUINT 2000 to compare our analysis with other static detectors.

### 6.2 Detecting Multiple Types of Faults

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

In Table 1, 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 as the number of statements where the constraint violations were found along some paths. We manually confirmed the data in the table.

Table 1: Detecting Multiple Types of Faults

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Size (kloc)</th>
<th>Buffer Overflow</th>
<th>Integer Fault</th>
<th>Null-Ptr Deref</th>
<th>Memory Leak</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$d$</td>
<td>$mf$</td>
<td>$fp$</td>
<td>$p$</td>
<td>$d$</td>
</tr>
<tr>
<td>wuftp1</td>
<td>0.2</td>
<td>4</td>
<td>0</td>
<td>1–11</td>
<td>0</td>
</tr>
<tr>
<td>sendmail6</td>
<td>0.2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>sendmail2</td>
<td>0.9</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1–4</td>
</tr>
<tr>
<td>polymorph-0.4.0</td>
<td>0.9</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1–4</td>
</tr>
<tr>
<td>gzip-1.2.4</td>
<td>5.1</td>
<td>10</td>
<td>0</td>
<td>2</td>
<td>1–35</td>
</tr>
<tr>
<td>tigervnc-1.2.2</td>
<td>45.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>ffmpeg-0.4.9pre</td>
<td>48.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>putty-0.36</td>
<td>60.1</td>
<td>7</td>
<td>2</td>
<td>1–15</td>
<td>4</td>
</tr>
<tr>
<td>apache-2.2.4</td>
<td>208.9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Under Buffer, we show a total of 33 buffer overflows with 5 false positives. We did 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 Marple, a manually constructed buffer overflow detector [17]. The results show that the generated detector is able to report all buffer overflows detected by Marple. 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 undetected infeasible paths, 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 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., $p 
eq \text{NULL}$, 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 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 analyzing 1–4 procedures. The data from gzip and putty imply that although in general, the faults were discovered by only propagating through several procedures, we are able to identify faults deeply embedded in the program across the maximum of 35 procedures. Without path information, it is difficult for manual inspection to understand how such a fault is produced.

### 6.3 Scalability

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.

In Table 2, we first give the time used for preparing the fault detection, including building the ICUF, and conducting pointer analysis and infeasible path detection. We then list for each type of fault, the number of queries raised in Column $q$, and the time used for resolving them in Column $t$. The experiments show that all the benchmarks 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 138 minutes.

Our experimental results show that 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 do not 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.

### 6.4 Comparison with Memory Leak Detectors

In the second experiment, we compare our memory leak detection with three other tools using SPEC CPUINT 2000. The first two tools [6, 22] 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
Table 2: Scalability

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CPU (sec)</th>
<th>Apache Analysis</th>
<th>MO_06</th>
<th>SC_07</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>wuftp1</td>
<td>10.1 s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sendmail1</td>
<td>25.0 s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>polymorph</td>
<td>6.4 s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gprof</td>
<td>11.2 s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tsieve</td>
<td>21.9 s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ftp@1.7 m</td>
<td>25.0 s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>apache</td>
<td>72.8 m</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

is discovered if the collected information contradicts the assump-
tion [22]. SC_07 converts the detection for memory leak to a reach-
ability problem using a guarded value flow graph [6]. 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 [7]. 

Table 3: Comparison on Memory Leak Detection

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size (kloc)</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>181.mcl</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>256.bzip2</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>197.parser</td>
<td>6.4</td>
<td>0</td>
</tr>
<tr>
<td>175.vpr</td>
<td>9.6</td>
<td>0</td>
</tr>
<tr>
<td>164.gzip</td>
<td>10.0</td>
<td></td>
</tr>
<tr>
<td>186.crafty</td>
<td>11.3</td>
<td></td>
</tr>
<tr>
<td>300.twolf</td>
<td>15.1</td>
<td>0</td>
</tr>
<tr>
<td>252.com</td>
<td>19.3</td>
<td>1</td>
</tr>
<tr>
<td>254.gap</td>
<td>31.2</td>
<td>1</td>
</tr>
<tr>
<td>255.vortex</td>
<td>44.3</td>
<td>1</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>64.5</td>
<td>1</td>
</tr>
<tr>
<td>176.gcc</td>
<td>128.4</td>
<td>2</td>
</tr>
</tbody>
</table>

The results are shown in Table 3. The data for other three de-
tectors are directly taken from papers [6, 7, 22]. 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 num-
bers 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 indi-
cate 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; 
evertheless, 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 it does not. For SC_07, our intuition is that besides 
using the guards on the value flow graph to help precision, other 
heuristics 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 are able to terminate the analysis early.

6.5 Comparison with Saturn

Our third experiment compares the analysis produced by Athena 
with Saturn 1.2 [26] on detecting null-pointer dereferences. As 
Athena runs on Windows while Saturn runs on UNIX, we 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 Ta-
ble 4 under Athena Analysis 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 
p ath detection is done, and time lists the performance of Saturn.

Table 4: Comparison on Null-Pointer Dereference Detection

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size (kloc)</th>
<th>Athena Analysis</th>
<th>MO_06</th>
<th>SC_07</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>181.mcl</td>
<td>1.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>2.9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>197.parser</td>
<td>6.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>175.vpr</td>
<td>9.6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>164.gzip</td>
<td>10.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>186.crafty</td>
<td>11.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>300.twolf</td>
<td>15.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>252.com</td>
<td>19.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>254.gap</td>
<td>31.2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>255.vortex</td>
<td>44.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>64.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>176.gcc</td>
<td>128.4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4 shows that our analysis terminates on 9 out of 12 bench-
marks, and it runs out of memory on the three largest benchmarks, 
indicated by EOM under ptr_time. Among the 9 programs that fin-
ish, 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 be-
cause it is written in C++. Saturn reports a total of 86 warnings, 
among the 51 we inspected (we selected the first 10 warnings for perlbmk), 7 are confirmed as null-pointer dereferences. Our inspec-
tion 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 interpro-
cedural 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 impre-
cision 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 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 Sa-
turn. 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 finding memory leaks.

6.6 Limitations

Athena is a research prototype and still has room for improvement. For example, we use Phoenix to build the ICFG, which currently does not model certain control flow such as function pointers and virtual functions. Therefore, certain parts of the code in a benchmark cannot be analyzed. Also, for detecting buffer overflow and memory leak, we only identify a set but not all of the code signatures where a query can be raised for checking safety. As an example, we do not construct queries at realloc for memory leak. Similarly, our infeasible path detection only handles certain types of conditional branches, but not all.

7. RELATED WORK

Much research has been done for static fault detection due to its importance. The uniqueness of our framework is that we generate demand-driven analyses, and report path-segments for both control- and data-centric faults. In Table 5, we show the three commonly applied static techniques in the state-of-the-art. Type based tools detect faults by enforcing the type safety on C. Examples include Rich [3], CQual [9] and CCured [21]. 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 [5] and Java PathFinder [24], 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 [11] and FindBugs [12] are representative path-insensitive dataflow analysis tools.

Path-sensitive tools are summarized in Table 6 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 this paper. Compared with the other six path-sensitive tools, Athena is the only one that handles integer faults. Except for Archer [27], 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 [14] and ESP [8] provide finite automata for modeling the faults. Prefix [4] develops a way to specify library calls. Saturn [26] 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 Athena does.

We also compare the tools in Table 6 with regard to the way paths are traversed in the analysis. The comparison on the three metrics show that Athena is different from the other tools in that we apply a demand-driven algorithm, which allows us to explore only relevant paths, instead of exhaustively exploring 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 Athena 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 bit-accurate modeling accomplished by Saturn and Calysto, we are not able to handle integer bit operations; however, the trade-off is a faster analysis.

To report an error, Prefix, ESP and Athena give path information for diagnosis. Athena provides the path segments where the root causes of a fault are located. Although the fault detection is path-sensitive, other tools only report a statement where a fault occurs.

Our work benefits from program slicing [25] in that we only look for program statements on which the demands (i.e., queries used to determine faults) depend. 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 [16], dataflow analysis [10], infeasible path detection [2] and bug detection [17, 19]. The demand-driven analysis can be path-sensitive [17] or path-insensitive [10], forward [19] or backward [2]. 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.

8. CONCLUSION

In this paper, we present a unifying framework, which includes a generator for automatically producing desired fault detectors, a general, scalable, static analysis, and a specification technique. The generated analyses are path-sensitive and interprocedural, and return path segments where a fault occurs. Our experiments show that the automatically produced analysis can identify buffer overflows, integer faults, null-pointer dereferences and memory leaks. Although in this paper 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.

9. REFERENCES

Table 5: Static Techniques for Identifying Faults

<table>
<thead>
<tr>
<th></th>
<th>type based</th>
<th>model checking</th>
<th>dataflow analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>path-sensitive</td>
<td>MOPS [5], JavaPathFinder [24]</td>
<td>see Table 6</td>
<td>Splint [11], FindBugs [12]</td>
</tr>
<tr>
<td>path-insensitive</td>
<td>Rich [3], CQual [9], CCured [21]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Path-Sensitive Dataflow Analysis for Identifying Faults

<table>
<thead>
<tr>
<th>Tools</th>
<th>Types of Faults</th>
<th>Specification</th>
<th>Path Traversal</th>
<th>Precision</th>
<th>Error Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>buf</td>
<td>int</td>
<td>control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metal [14]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>automata</td>
<td>×</td>
</tr>
<tr>
<td>ESP [8]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>automata</td>
<td>×</td>
</tr>
<tr>
<td>Archer [27]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>no</td>
<td>×</td>
</tr>
<tr>
<td>Saturn [26]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>summary</td>
<td>×</td>
</tr>
<tr>
<td>Calysto [1]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>no</td>
<td>×</td>
</tr>
<tr>
<td>Athena</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>assertions flow-funcs</td>
<td>×</td>
</tr>
</tbody>
</table>


