General, Scalable Path-Sensitive Fault Detection

Wei Le
Department of Computer Science
University of Virginia
weile@virginia.edu

Mary Lou Soffa
Department of Computer Science
University of Virginia
soffa@virginia.edu

ABSTRACT

Recent research has shown that program paths are important in static analysis for detecting and reporting faults. However, developing a scalable path-sensitive analysis is challenging. Often the techniques only address one particular type of fault and require much manual effort to tune for the desirable scalability and precision. In this paper, we present a novel framework that automatically generates scalable, interprocedural, path-sensitive analyses to detect user specified faults. The framework consists of a specification technique for expressing program properties related to faults, a scalable path-sensitive algorithm, and a generator that unifies the two. The generated analysis identifies not only faults but also the path segments that are relevant to the faults. The generality of the framework is accomplished for both data and control centric faults, so that the detection of multiple types of faults can be unified, which enables the exploitation of fault interactions for diagnosis and efficiency. We implemented our framework and generated fault detectors for identifying buffer overflow, integer truncation and signedness errors, and null-pointer dereference. We experimentally demonstrated that the generated analysis scales up to at least half a million lines of code, and its detection capability is comparable to manually produced analyses. In our experiment, a total of 53 faults of the three types from 9 benchmarks are detected, among which 37 have not been reported previously. The results show that we are able to identify faults deeply embedded in the code, and the average length of faulty path segments is 1–4 procedures, which provides a focus for diagnosis.

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. At Microsoft, static detectors report thousands of faults monthly [16]. Static detectors are deployed on desktops for software developers, and code is only allowed to check in after it passes the inspection of those tools. Research has shown that program path information is essential in static analysis for achieving precise fault detection, especially for complex faults [5, 14, 20]. In path-sensitive analysis, identified infeasible paths can be excluded, and program facts collected along different paths for determining faults are never merged, avoiding a loss of precision. Furthermore, because a fault often results from a sequence of executions, reporting a fault in terms of path segments instead of program points allows the code inspector to follow the actual code sequences that lead to the fault.

Although useful for both detecting and reporting a fault, developing a path-sensitive detector that is scalable for various types of faults is challenging. In current path-sensitive detectors, scalability is mostly addressed for a certain type of fault. For example, ESP [5] achieves a polynomial time verification on identifying typestate violations. The scalability is achieved by applying heuristics to select information relevant to the fault. The approach is only shown effective for typestate errors because its underlying assumption is based on how a programmer might program typestate transitions in practice. Tools such as Saturn [20] apply a modular, instead of whole program analysis. That is, parts of program are analyzed in isolation and then the results are composed at the whole program level. This summary based technique is not effective for faults whose detection requires a large amount of summary information or the tracking of global side effects. In fact, Saturn only demonstrates its effectiveness in detecting typestate violations. Besides heuristics and modular analysis, applying annotations is also a solution for scalability. ESPx [10] annotates procedural interfaces, aiming to convert an interprocedural analysis to intraprocedural. However, supporting multiple types of faults increases the use of annotations while both writing and verifying annotations require much human effort. ESPx only supports buffer related contracts. Our previous research developed a demand-driven analysis to achieve the scalability in detecting only buffer overflow [14].

Considering the manual effort and challenges to build individual fault detectors, techniques that address scalability and also are applicable to a variety of faults are desirable. Because of the generality, such a system can enable the detection of interactions of different types of faults, such as showing an integer fault can lead to buffer overflow.

In this paper, we present a novel framework which enables the automatic generation of scalable path-sensitive analyses that detect user specified faults. The framework consists of

a general, scalable algorithm, a specification technique, and a generator that unifies the two. The key idea is to address the scalability of path-sensitive fault detection in a general demand-driven algorithm, and automatically generate the fault-specific part of the analysis from a specification.

The generated analysis is path-sensitive in that program facts collected along different paths are distinguished. Statically identifiable infeasible paths are also excluded. Importantly, only path segments that are relevant to the production of faults are reported, providing both the focus and necessary context for the code reviewer to diagnose the faults. The scalability of path-sensitive analysis is achieved by a demand-driven algorithm and a set of optimizations, all of which are encapsulated in the general algorithm.

The generality of the framework is accomplished for both data and control centric faults. Data centric faults require the tracking of variable values and ranges for detection, e.g., buffer overflow and integer faults, while control centric faults, such as typestate violations, mainly focus on the order of operations. Although different types of information are required, there are commonalities in fault detection. Our insight is that 1) many types of faults are only observable at certain types of program statements, and 2) on the paths to such observable points, only certain types of statements can contribute to the failure. By identifying such observable points, we can construct a query at those points regarding whether the fault can occur and propagate the query along the paths for resolution. Similarly, given contributing points, we know where to collect information to resolve the query and thus determine the fault. That is, by supplying observable/contributing points and the corresponding actions at the points, we are able to guide a general analysis to locate the desired faults.

We develop a specification language for users to specify such information. The specification uses code signatures to identify observable and contributing points, and introduces the basic construct *attributes* to map the code to desired abstractions. For example, attribute `Len(s)` represents the length of the string `s` and `Value(i)` expresses the integer value stored in the variable `i`. Using attributes and a set of operations on the attributes, we specify the actions at observable/contributing points. To generate an analysis, the specification is translated into code modules, and plugged into a backward demand-driven symbolic simulator to produce desired fault detectors.

We implemented our framework using the Microsoft Phoenix infrastructure [18], and the Disolver constraint solver [12]. We experimentally show that our framework is able to generate fault detectors whose detection capability is comparable with manually constructed ones. We show that the generated analyses are scalable to half a million lines of code for detecting common faults of buffer overflow, integer error and null-pointer dereference with path-sensitive precision. The experiments also demonstrate that while we are able to discover faults deeply nested in 7 procedures, the path-segments for faults are generally short, 1–4 procedures on average.

To the best of knowledge, our framework is the first that can automatically generate interprocedual path-sensitive fault detectors which scale for both data and control centric faults. Our contributions include:

- a framework that enables automatic generation of fault detectors,
- a scalable algorithm that detects both control and data centric faults and returns path segments of the faults,
- a specification language for user specified faults, and
- implementation and experimentation that demonstrate the generality and scalability of the framework.

In the rest of the paper, we present the three components of the framework: a general analysis in Section 2, specification in Section 3, and the generator in Section 4. We show the experimental results in Section 5, and compare the related work in Section 6. The conclusion is given in Section 7.

2. SCALABLE ANALYSIS

Demand-driven is a static analysis technique that determines where to collect the information from the program source, what information to collect, and when the analysis should terminate. A demand-driven analysis detects faults by raising queries at the statements where faults potentially occur, inquiring whether the constraints of correctness would hold. The analysis performs a backward path traversal to determine the occurrence of the fault. In this section, we first use an example to intuitively explain how a demand-driven analysis works to identify different types of faults. We then define key components in a general analysis, and present a general algorithm developed based on the definitions.

2.1 An Example for Demand-Driven Analysis

In Figure 1, an integer signedness error occurs at node 11. We show how our analysis identifies this fault. In the first step, the analysis 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. Path segment $\langle 6, 8, 10, 11\rangle$ is reported as faulty, and nodes 6 and 10 are highlighted on the path, as they contribute to the query update and are likely helpful to understand the fault.

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. Our approach is that we run a demand-driven branch correlation analysis before the fault detection and mark the infeasible paths on the interprocedural control flow graph (ICFG) of the program [1]. In Figure 1, we show two infeasible paths that are relevant to null-pointer dereference detection, denoted as $\texttt{ip}_1$ and $\texttt{ip}_2$. 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 the infeasible path

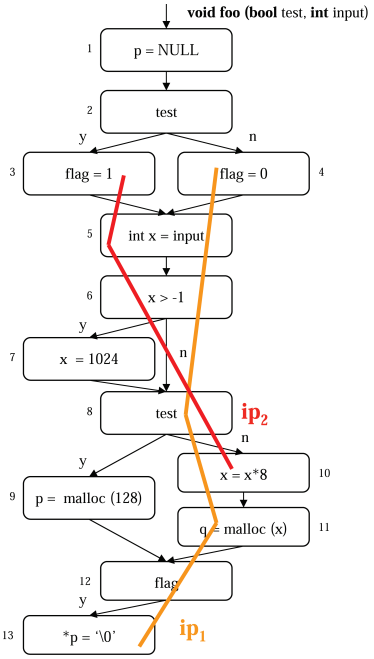

Figure 1: Detecting Different Types of Faults

and records $\texttt{ip}_1$ in progress. Along one path $\langle 13, 12, 9\rangle$, the propagation no longer follows the infeasible path and thus the query drops $\texttt{ip}_1$. The query is resolved as safe at node 9 as `malloc` implies a non-NULL `p`, assuming memory allocation successes. Along the other path $\langle 13 - 10\rangle$, no update occurs until the end of $\texttt{ip}_2$ is met at node 10. The query thus records $\texttt{ip}_2$ in progress. When the query arrives at branch $\langle 5, 4\rangle$, the start of $\texttt{ip}_1$ is discovered, showing the query traverses an infeasible path. The analysis terminates. Similarly, the propagation halts at branch $\langle 5, 3\rangle$ for traversal of $\texttt{ip}_2$. The analysis reports node 13 as safe for null-pointer dereference.

2.2 Components in a General Fault Detection

In this section, we show why a general algorithm can be developed to generate individual fault detectors. We reveal the commonality among fault detection, and identify components in analysis that are parameterizable for different faults.

Definition 1: A *path* is a sequence of statements in a program, starting at the entry of the program, and ending at the exit. A *path segment* is any subsequence of statements on the path. An input exercises a path, producing an *execution*. If no input can be found to exercise the path, the path is *infeasible*.

Definition 2: A *property* α for a path p is a set of constraints on a sequence of program states along the execution of p. α *holds* for p if and only if all executions of p satisfy the constraints, while α is *violated* if and only if there exists an execution of p that does not satisfy the constraints.

Constraints define conditions on abstractions of program states. The abstraction extracts the attributes of program objects such as value, range, or typestate of individual program variables, or relations of multiple variables.

For example, "along any path execution, a write to a file only can be performed after the file is open" is a property. The constraint requires when file write is invoked, the typestate of file is always open.

Definition 3: If there exists a statement s along p, at which the violation of α can occur, we say property α is *observable* at s, and s is an *α-point*. The constraints used to determine the property at α-points are called *α-constraints*.

For example, a buffer overflow can be produced after executing a sequence of statements along a path. However, only when the execution reaches a buffer write statement, can the occurrence of the buffer overflow be confirmed. Thus the buffer write statement is an α-point. In the previous file open/write example, file write is an α-point. In Figure 1, node 11 is an α-point for integer violation, and the corresponding α-constraint is $[\texttt{Value(x)}\geq 0]$.

Definition 4: *α-impact* is a statement on the path that can contribute to the determination of α-constraints. *α-transition* is the change of the program state at the α-impact.

Intuitively, α-impacts are a slice of statements along the path that determines the outcome of the α-constraints. α-transitions instruct the change of the data facts at α-impacts, and are similar to transfer functions in data flow analysis.

In Figure 1, we report the faulty path $\langle 6, 8, 10, 11\rangle$ for integer errors, and nodes 6 and 10 on the path are α-impacts. The α-transition at node 6 is a symbolic substitution, and the α-transition at node 10 integrates the constraint from a conditional branch.

Definition 5: A path segment between the first α-impact and α-point is *α-segment*.

An α-segment contains a complete sequence of α-impacts. The transitions of program states at those α-impacts determine the α-constraints of the property at α-point. In Figure 1, path $\langle 6, 8, 10, 11\rangle$ is an α-segment, and node 6 is the first α-impact. Path $\langle 9, 12, 13\rangle$ is also an α-segment.

The goal of the fault detection is to determine whether a given property α can be violated. Our hypothesis is that if the four α-elements (α-points, α-constraints, α-impacts and α-transitions) are identifiable for a property, we are able to compute α-segments for the faults. The common faults of buffer overflow, integer error, and null-pointer dereference, shown as above, all belong to this category.

2.3 A General Algorithm

Identifying four α-elements as dependent on the faults, we apply a demand-driven analysis to compute α-segments. Previous research has demonstrated the scalability of demand-driven analysis in data flow computation, infeasible path identification, and pointer analysis [2, 7, 13]. In this paper, we aim to show that a demand-driven analysis can also make path-sensitive fault detection scalable and applicable for a variety of faults.

In Figure 2, we show a general demand-driven algorithm and its interaction with the four α-elements for the discovery of faulty α-segments. The analysis first scans the program to identify α-points in the program. α-constraints are used to construct the query. The query is propagated backwards in a path-sensitive fashion. When an α-impact is encountered, the analysis updates the query accordingly. For every update, the query is evaluated as to whether the α-constraints are always true or can be false. The analysis terminates if the resolutions of the query are derived. If the query is resolved as unsafe, path segments where the query propagates are faulty α-segments. A forward traversal can report all faulty α-segments.

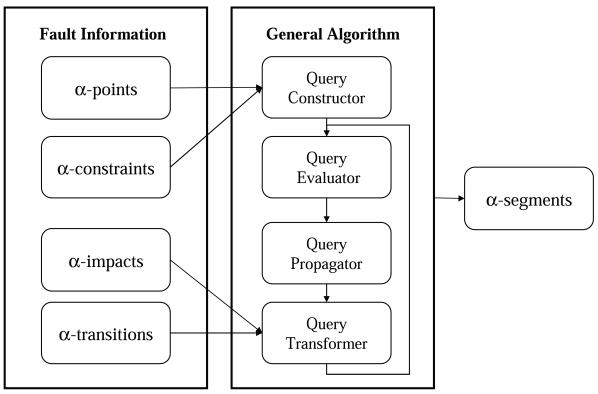

Figure 2: Interaction of α-elements and analysis

Vars		*Vbuffer* `a`, `b`; *Vint* `d`; *Vany* e;
ObservablePoints		
	α-point	`$strcpy(a,b)$`
	α-constraint	Size(a)≥Len(b)
or		
	α-point	`$memcpy(a,b,d)$`
	α-constraint	Size(a)≥min(Len(b), Value(d))
or		
	α-point	`$a[d]=e$`
	α-constraint	Size(a)>Value(d)
ContributingPoints		
	α-impact	`$strcpy(a,b)$`
	α-transition	Len(`a`) := Len(`b`)
or		
	α-impact	`$strcat(a,b)$`
	α-transition	Len(`a`) := Len(a)+Len(`b`)
or		
	α-impact	`$a[d]=e$` && Value(e)='\0'
	α-transition	(Len(`a`)>Value(`d`) ‖ Len(`a`)=`undef`) ↦ Len(`a`) := Value(`d`)

Figure 3: Snapshot of Buffer Overflow Specification

2.4 Challenges of Analysis and Optimizations

The challenges for designing a general algorithm are 1) the strategies of query propagation, i.e., how the queries traverse branches, procedures or loops, and 2) the solutions for handling imprecision sources, such as library calls or pointers. Here, we present our approaches to address the challenges encountered in a typical static path-sensitive analysis.

Propagating through branches, the query is copied at the fork point, each of which is advanced into separate branches. At the branch merge point, queries from different branches continue to propagate along the paths.

Our interprocedural analysis is context-sensitive and also considers global side effects. When a query arrives at a procedural call, 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. Upon reaching the beginning of the procedure, the query is propagated to the caller where it originally comes to preserve the context sensitivity.

We classify loops into three types based on the update of the query in the loop. We propagate the query into the loop 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). If the query is still not resolved, we introduce a "don't-know" tag to record the imprecision. 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. In this way, targeted heuristics can be introduced with controllable imprecision, or other compensable techniques such as testing or statistical inference, can be applied to address the don't-know that static analysis is not powerful enough to resolve.

We also report "don't-know" for resolving other imprecision sources such as external libraries or non-linear operators. By inspecting the don't-knows generated via our initial analysis, we identify which libraries or non-linear operators block the analysis, and manually model them. To handle the C structure and heap, we apply an external pointer analysis, which is intraprocedual, flow-sensitive and field-sensitive.

To further improve the efficiency, we develop optimizations that are generally applicable. The goal of the optimizations is to increase the reusability of the intermediate results. For example, we cache the queries at the nodes so that when the same query arrives later, it can directly use the cached results.

3. SPECIFICATION

In the previous section, we identified the four α-elements that can be used by a general demand-driven analysis to detect desired faults. In this section, we present a specification language that allows the users to specify α-elements of a property.

3.1 An Overview and Key Insights

In a specification, we describe a property using two aspects: 1) we present a list of α-points and α-constraints to show where the property can be observed; and 2) we provide a set of α-impact and α-transition pairs to project how the property is effected along the paths in a program. The α-points and α-impacts are given in terms of statement types. During analysis, if a particular statement in the program matches the given type listed as α-points or α-impacts, actions specified at α-constraints/transitions will be instantiated.

For the above requirements, our specification language includes the following design. To specify types of statements required by α-points/impacts, we use code signatures with conditions on the statements. To express α-constraints, Boolean operators and comparators are introduced. For α-transitions, we add operators to specify actions such as symbolic substitution or integration of additional constraints. We also support conditional actions for α-constraints/transitions that are relevant to the context of the program. Importantly, a fundamental construct, namely an *attribute*, is developed to specify the abstractions of program objects. A set of attributes are composed using arithmetic, Boolean or action operators to compose α-constraints/transitions.

S → **Vars** VarList **ObservablePoints** ObsList
ContributingPoints ConList
VarList → Var*
Var → VarType *namelist*;
VarType → *Vbuffer*|*Vint*|*Vany*|*Vptr*|...
ObsList → Obs ⟨**or** Obs⟩*
ConList → Con ⟨**or** Con⟩*
Obs → α-**point** Point α-**constraint** Condition
Con → α-**impact** Point α-**transition** Command
Point → $LangSyntax$|Condition|$LangSyntax$&&Condition
Condition → Attr Comparator Attr|!Condition|[Condition]|
Condition&&Condition|Condition || Condition
Command → Attr:=Attr| ∧ Condition|Condition ↦ Command|
Command&&Command|Command || Command|[Command]
Attr → PrimAttr(*var*, ...)|Constant|Attr Op Attr|*min*(Attr,Attr)|
[Attr,Attr]|¬Attr|!Attr|Attr ∘ Attr|[Attr]
PrimAttr → *Size*|*Len*|*Value*|*MatchOperand*|*TMax*|*TMin*|...
Constant → *0*|*true*|*false*|*undef*|...
Comparator →= | ≠ | > | < | ≥ | ≤ | ∈ | ∉
Op → +| − | ∗ | ∪ |∩

Figure 4: Grammar for Specification Language

As an example, we show a snapshot of specification for detecting buffer overflow. In Figure 3, the specification consists of three sections. Keyword `Vars` defines a collection of variables used in the specification. Under **ObservablePoints**, we provide a list of α-points and the corresponding α-constraints. The snapshot gives three examples of α-point, including the library calls of `strcpy` and `memcpy` as well as the direct assignment to the buffer. Symbol "$" includes the code signature for a particular type of statement. The role of variables such as `a` and `b` is to communicate operands between α-point and α-constraint. `Size(a)` and `Len(b)` are attributes, representing the size of the buffer and the length of the string in the buffer respectively. We use a comparator "≥" on attributes of `Size(a)` and `Len(b)` to specify the buffer safety constraint. The section under **ContributingPoints** provides α-impacts and α-Transitions. Consider the first pair in Figure 3. It says 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 α-transition using symbol ↦. It says when an '\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`.

3.2 Grammar and Semantics

We develop a specification language to express the four α-elements. In Figure 4, we show how a set of advanced language constructs can be composed from the basic construct of attributes. In the grammar, terminals are highlighted: keywords use bold fonts, and the predefined constants, functions and types are italicized.

We show in the first rule, that a specification consists of three sections. In the first section, we define *specification variable*. The specification variables represent program objects that can be used to describe the four α elements, such as statements and operands. The rule of `Var` shows a variable is defined by a type and a name. A set of built-in types are 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 that corresponds to a program variable has a type starting with a `V`, followed by a name indicating the type of program variable such as `int`.

Vars		*Vint* `x,y,z,d`; *Vbuffer* `b`;
ObservablePoints		
	α-point	`$x=y+z$`
	α-constraint	(Value(`y`)+Value(`z`))∈[TMin(`x`),TMax(`x`)]
or		
	α-point	`$x=y-z$`
	α-constraint	(Value(`y`)-Value(`z`))∈[TMin(`x`),TMax(`x`)]
or		
	α-point	`$x=y*z$`
	α-constraint	(Value(`y`)*Value(`z`))∈[TMin(`x`),TMax(`x`)]
ContributingPoints		
	α-impact	`$x=y$`
	α-transition	Value(`x`) := Value(`y`)
or		
	α-impact	`$d=strlen(b)$`
	α-transition	Value(`d`) := Len(`b`)
or		
	α-impact	`$if(y≥0)$`
	α-transition	∧ Value(`y`)≥0

Figure 5: Integer Overflow/Underflow

After specification variables, the grammar gives the productions of observable points, `ObsList`, and contributing points, `ConList`. `ObsList` consists of pairs of α-points and α-constraints, using the keyword **or** for multiple pairs. Similarly, `ConList` lists pairs of α-impacts and α-transitions. The construct `Point` provides code signatures or/and conditions about the statements to identify types of statements required by α-point and α-impact. The production `Condition` compose conditions of attributes, e.g., used as α-constraint. The basic rule is to connect two `Attr` with a `Comparator`. A condition is a Boolean. Therefore, a set of Boolean operators can be applied. Symbol [] is used to define the priority of the computation. The construct `Command` specifies the actions that can be taken on attributes with the operators of `:=` for assignment and ∧ for integrating conditions. For conditional actions, we introduce `Condition` to compose `Command`, using operator ↦. The construct `Attr` used in `Condition` and `Command` specifies the attributes of variables. In our specification language, we define a set of commonly used primitive attributes as terminals, shown in the `PrimAttr` production. A set of operators are defined to compose attributes. See the productions `Attr` and `Op`.

3.3 Specification Examples

In this section, we show two other specifications, one for a data centric fault and the other for a control centric fault.

Figure 5 provides a specification for identifying integer overflow/underflow. The specification defines α-points as statements that perform integer arithmetics. In the figure, we give examples of plus, minus and multiply. α-constraints require that the result of integer arithmetic should fall in the range that the destination integer can store, specified

Vars		*Vptr* a,b, c; *Vint* d; *Vany* e;
ObservablePoints		
	α-point	`MatchOperand(*a, a->e, a[d])` $\neq \emptyset$
	α-constraint	Value(a)$\neq 0$
ContributingPoints		
	α-impact	$a=c$&&IsConstStr(c)
	α-transition	$\wedge$ Value(a) $\neq 0$
or		
	α-impact	$a=NULL$
	α-transition	Value(a):= 0
or		
	α-impact	$a=b$
	α-transition	Value(a):= Value(b)

Figure 6: Null-pointer Dereference

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. Therefore, statements that potentially update the value or range of an integer are identified, as three examples show in the figure.

In the second example, we show a specification for detecting null-pointer dereference. In Figure 6, α-point presents an attribute, MatchOperand, which examines operands in the statement to match the pointer dereference signatures of *a, a->e, or a[d]. The α-constraint shows that we should construct a query as to whether the pointer is a non-NULL. The first two pairs of α-impacts and α-transitions imply that when a NULL or a constant string is assigned to the pointer, the query is resolved. The third rule specifies a symbolic substitution rule between two pointers.

3.4 Usability of the Specification

The components of a specification can be reused. Although for different types of faults, α-points/constraints are different, α-impacts/transitions can be common. For example, detecting both integer faults and buffer overflows requires tracking the values of integers or the lengths of the strings. In fact, in our experiments, we use the same set of α-impacts/transitions and successfully identify integer faults and buffer overflows. In our experience, the length of the specification for each property is about a half to one page, and most of the α-impacts/transitions are used to resolve don't-know warnings such as documenting semantics of library calls. To extend the specification language to support a new type of fault, in the worst case, we need to introduce new primitive attributes and maybe operators. The generality is achieved based on the assumption that while the required abstractions in the analysis, i.e., attributes, are limited to certain types, it is the composition of the attributes that define different queries and transfer functions, and thus determine a special purpose fault detector.

4. GENERATING ANALYSIS

Two steps are required to generate an analysis to detect specified faults. We first parse the specification and generate the code for the query constructor and transformer (see Figure 2 for the components in a general algorithm); we then plug in these modules into the query propagator to generate the desired analysis.

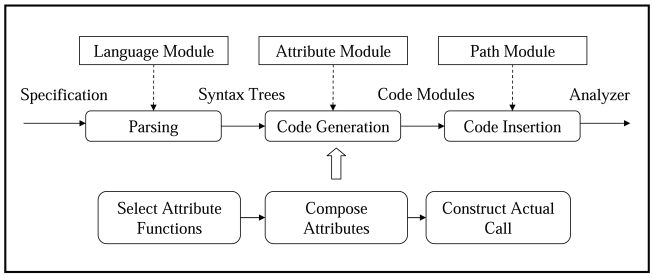

Figure 7: Flow of Generation

4.1 An Overview of the Approach

The generator consists of three basic modules: the *language module*, the *attribute module* and the *path module*. The language module implements the grammar and semantics. The attribute module encapsulates a repository of attribute definitions. The path module implements the general demand-driven algorithm presented in Section 2.

The generator, shown in Figure 7, works similarly to a compiler. At the first step, we parse the specification. A preprocessor is developed to convert the specification into an intermediate representation, where the code signature encapsulated in symbol $ is replaced with a set of conditions with regard to the operands and operator of the statement. After code signatures are processed, a stream of conditions and commands generated from the preprocessor are parsed into a set of syntax trees. A syntax tree represents an α-point, α-constraint, α-impact or α-transition. Trees are paired for α-point/constraint, and for α-impact/transition. As an example, we show in Figure 8 the parsing process for the first pair of α-point and α-constraint in buffer overflow specification. To covert the code signature, the preprocessor introduces the attribute of Op(s) to represent the operator of statement s, and $\texttt{Src}_i\texttt{(s)}$ for the i^{th} operands. Specification variables a and b are replaced accordingly. The parser generates trees A and B. Tree A corresponds to the α-point and tree B is translated from α-constraint. The linkage between A and B defines they are a pair in the specification.

After syntax trees are produced, we take one syntax tree at a time, and generate code for the corresponding α element. In Figure 7, we show three steps to generate code for a syntax tree. As each node in the tree corresponds to a primitive attribute, we first select the definitions of individual primitive attributes from the attribute repository. For example, for tree A in Figure 8, we need to find the call that implements the Op attribute in the attribute module. Box 1 at the bottom of the figure gives a snapshot of the code. After attributes are selected, we compose them based on the structure of the syntax tree, using the semantics of operators defined on the attributes, such as computation, comparison or composition. For instance, in Box 2 in Figure 8, we integrate the call Op and then generate code that interprets the comparison symbol =. Finally, we generate actual calls for a syntax tree. As shown in Box 3 in Figure 8, we supply the actual parameter s and generate isnode = treeA(s). This code generation step outputs a list of code modules, each of which implements the semantics of a syntax tree. The

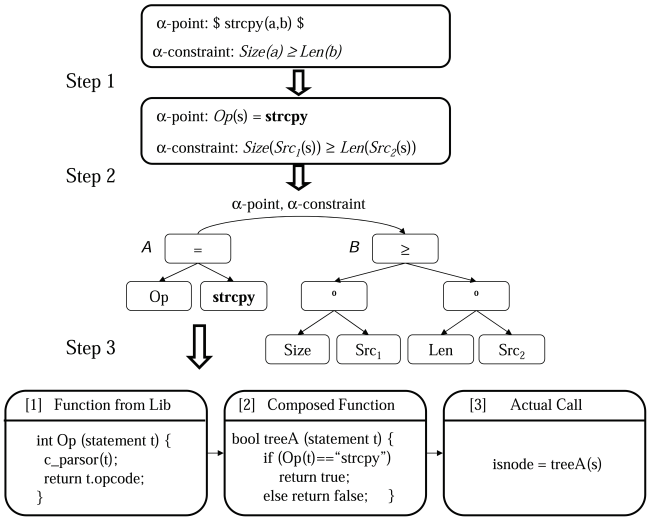

Figure 8: Parsing Specification into Syntax Trees

code modules are paired if they implement a corresponding α-point and α-constraint, or α-impact and α-transition.

In the last step, we integrate those code modules into a general demand-driven algorithm. Based on the relationships marked on the links of syntax trees, we are able to distribute them to the right "holes" in the analysis.

4.2 The Algorithm for Generating Analysis

Algorithm 1 shows in detail the process of generating fault detectors. It takes a user provided specification and outputs the source code of a desired analysis. In the algorithm we use a global variable `A` to store the code of the analyzer as the generation goes. We introduce `attr` to denote the attribute repository and `l` for the language module. The algorithm implements three general tasks. Line 2 parses the specification. Lines 3 to 16 process syntax trees and generate the code that implements the semantics of the trees. Lines 17-18 insert code to a demand-driven template and produce the analysis.

At line 1, we first generate the code that builds the ICFG and initialize a worklist. At line 2, we parse a given specification `Xspec` based on the language syntax `l.grammar`, obtaining a list of syntax trees in `treelist`. Trees are paired for corresponding α-point/constraint, and α-impact/transition. At line 4, we generate the code for the first syntax tree in the pair. The code generation for a tree takes three steps, as shown in function `CodeGenforTree` at line 19. `Select` at line 20 returns a list of functions that implement each attribute included in the tree. These functions are composed at line 21. The composed function is added into the analyzer at line 22. Based on the function signature, `ActualCall` at line 23 supplies a set of actual parameters through `arglist` to produce a concrete call. Note that `arglist` is chosen in a way that the parameters for all generated calls can be integrated into the demand-driven template. At line 4, the actual call that selects a particular program point for either α-point or α-impact is returned as *isnode*, while its body of implementation is integrated earlier to the analyzer at line 22. After *iscode* is generated, the algorithm determines if the next syntax tree in the pair is an α-constraint or α-

```
   Input  : Specification of Property X(Xspec)
   Output: Analyzer for the property (A)
 1 A = "icfg = BuildICFG(p); set worklist L to {};"
 2 treelist = Parse(l.grammar, Xspec)
 3 foreach pair ∈ treelist do
 4     isnode = CodeGenforTree (pair.first, "s")
 5     if α-Point (pair.first) then
 6         raiseQ =CodeGenforTree (pair.next, "s")
 7         code = "If isnode then q=raiseQ;
 8                 add (q,s) to L"
 9         add code to list1
10     end
11     else if α-Impact (pair.first) then
12         transQ =CodeGenforTree (pair.next, "s", "q")
13         code = "If isnode then transQ;"
14         add code to list2
15     end
16 end
17 GenerateRaiseQ (list1)
18 GenerateSolveQ (list2)
19 Procedure CodeGenforTree (tree t, arglist p1, p2...)
20 flist = Select (t, attr)
21 ftree= Compose (flist, t, l.semantics)
22 AddFunc (A, ftree)
23 return ActualCall (ftree, p1, p2,...)
24 Procedure GenerateRaiseQ (codelist list)
25 Append (A, "foreach s ∈ icfg")
26 foreach item ∈ list do Append (A, item)
27 Procedure GenerateSolveQ (codelist list)
28 Append (A, "while L ≠ ∅ {remove (q,s) from L; ")
29 foreach item ∈ list do Append (A, item)
30 Append (A, "if(q.resolved) add(q,s) to A[q]; else
   foreach(n ∈ Pred(s)) do Propagate(s,n,q);}")
```

Algorithm 1: Generating Analysis

transition, as different parameters are required to generate actual calls for the two. See lines 6 and 12. Calls of *raiseQ* and *transQ* are generated using the similar approach.

The generated code for a pair of syntax tree is added to a list at lines 9 and 14, which are then consumed at lines 17 to 18. Code for raising a query is generated at lines 24-26. At line 25, we first generate the code that enumerates each statement on ICFG. We then insert the code produced at lines 7-8 that can determine an α-point and construct a query (see line 26). The code generation for resolving a query is shown at line 27. For the code that is independent of the property under tracking, we directly integrate it to the analyzer `A` at lines 28 and 30. The part of analysis that is relevant to the property is included at line 29, where α-impact and α-transition are determined for each pair of `(q,s)` from worklist `L`. We use the code generated at line 13 to parameterize this part of the analysis.

4.3 Generated Analysis

The generated analysis (stored in `A` in Algorithm 1) is listed in Algorithm 2. It takes a program p and computes paths of property `X` specified in `Xspec`. In Algorithm 2, lines 3, 4 and 8 take the code generated from lines 2-16 in Algorithm 1. `X_isnode` is stored in variable `isnode` in Algorithm 1, `X_raiseQ` is from `raiseQ`, and `X_transQ` from `transQ`. These three calls implement the part of the analysis that are relevant to the property. The rest of Algorithm 2 implements a demand-driven template that can be shared across analyses. The algorithm works as follows. First, at lines 2–5, each statement in the ICFG is evaluated, and the

```
   Input  : program (p)
   Output: faults of property X
 1 icfg = BuildICFG(p); set worklist L to {};
 2 foreach s ∈ icfg do
 3 |   if X_isnode (s) then
 4 |   q = X_raiseQ (s); add (q,s) to L;
 5 end
 6 while L ≠ ∅ do
 7 |   remove (q, s) from L;
 8 |   if X_isnode (s) then X_transQ (s,q);
 9 |   if (q.resolved) then add(q,s) to A[q];
10 |   else
11 |   foreach n ∈ Pred(s) do PropagateQ(s,n,q);
12 end
```

Algorithm 2: Generated Analysis

query is raised if α-point is encountered. At line 8, the analysis updates and evaluates the query `q` if the statement `s` is determined to be an α-impact. At line 9, the resolution of the query is reached, and the analysis terminates for this query. The paths the query has been propagated to are presented as faulty α-segments. If the query is not resolved, at line 10 the analysis continues to propagate the query; a set of rules we designed for query propagation are implemented in `propagateQ` at line 11.

4.4 Using the Framework

Using our framework, the user can write a specification and generate an analysis for one type of fault. Also, the user can integrate α-elements for multiple types of faults in one specification, and generate an analysis for identifying multiple types of faults. In this case, feasibility information and cached queries can be shared across the detection of different faults. Meanwhile, the integration enables the exploration of interactions of distinct types of faults, e.g., determining if an integer fault can impact a buffer overflow. As analyzing for each α-point is independent, the analysis can be run in a parallel.

5. EXPERIMENTAL EVALUATION

To evaluate our work, we implemented our framework using the Phoenix compiler infrastructure and the Disolver constraint solver from Microsoft [12, 18]. Phoenix provides pointer analysis, ICFG construction, and a part of the attribute module. Disolver is used to evaluate integer constraints. The goals of the experiments are to demonstrate path-sensitive fault detectors can be automatically generated, and the generated detectors are scalable and comparable to manually constructed fault detectors. Our benchmark suite consists of 9 C/C++ programs, where `wuftp-1`, `sendmail-2` and `sendmail-6` are from Zitser et al. [22], and the rest are real-world deployed software. In the benchmarks, 8 programs contain bugs, which were previously reported by static analysis, runtime detection or manual inspection. They are useful for evaluating the false negative rates of our detectors. `apache-2.2.4`, with about half a million lines of code, is chosen for demonstrating the scalability of our detectors.

5.1 Detecting Faults

In the first experiment, we generate three detectors from our framework, which identify buffer overflow, integer truncation/signedness errors, and null-pointer dereference respectively. Infeasible path detection is integrated in the detectors. We demonstrate the effectiveness of the detectors using three metrics: detection capability, false positives and false negatives, shown in Table 1. Faults here are counted as the number of α-points where α-constraints are violated along some paths. We manually confirmed warnings for the data in the table.

Table 1: Fault Detection

Benchmarks	Buffer			Integer			Pointer		
	d	f_n	f_p	d	f_n	f_p	d	f_n	f_p
wuftp-1	4	0	0	0	0	0	0	0	0
sendmail-6	0	0	1	6	0	0	0	0	0
sendmail-2	4	0	0	0	0	0	0	0	0
polymophy-0.4.0	7	0	0	2	0	0	0	0	0
gzip-1.2.4	9	0	1	5	0	0	0	0	2
ffmpeg-0.4.8	0	0	0	4	0	0	1	2	6
tightvnc-1.2.2	0	0	1	4	0	0	0	0	2
putty-0.56	0	0	1	3	0	2	0	0	1
apache-2.2.4	1	0	1	2	0	3	1	0	4

Under *Buffer*, we show a total of 25 buffer overflows are reported with 0 false negative, and 5 false positives. 17 are newly discovered buffer overflows. The 5 false positives are all diagnosed as infeasible buffer overflow. As identification of infeasible paths is undecidable, we cannot exclude all infeasible paths statically. The four programs `wuftp-1`, `sendmail-2`, `polymophy` and `gzip` have also been used before to evaluate Marple, a manually constructed buffer overflow detector [14]. The results show that the generated detector is able to report all buffer overflows detected by Marple. Under *Integer*, we report a total of 26 detected integer faults with 0 false negative, among which 19 were not previously reported. The false positives are caused by insufficient infeasible path detection. The results for detecting null-pointer dereference are shown under *Pointer*. We identify 2 null-pointer dereferences, and 1 from `ffmpeg` was not reported before. We missed two faults in `ffmpeg` as they are related to interactions of integer faults, which we have not yet handled in our detectors. A total of 15 false positives are confirmed in detecting null-pointer dereferences, causing by imprecise pointer analysis and infeasible paths.

In summary, we identified a total of 53 faults of the three types from 9 benchmarks, and 37 are new faults that were not previously reported. Inspecting those new faults, we found that many of them are located along the same paths. As a result, the dynamic approaches would halt on the first fault but never trigger the rest. We missed 2 known faults and reported a total of 25 false positives for the detection, mainly due to the pointer analysis and infeasible path detection. Our experimental results demonstrate that the generated analyses are able to identify both control and data centric faults with reasonable false positives and false negatives. The results for buffer overflow detection shows that the capability of generated detectors are comparable with manually constructed ones.

5.2 Computing α-segments

Our generated analyzers not only detect faults, but also report useful path information for diagnosing the faults. Table 2 presents the results of α-segments and α-impacts for all identified faults. Column α-s reports the average length (in terms of procedures) of faulty α-segments before "/", and af-

ter "/" gives the maximum number of procedures the faulty α-segments cross. Column α-i shows the average number of α-impacts on the α-segments. The results show that, although the complete faulty paths can be very long, the α-segments we identify are only 1–4 procedures on average for identified faults, and only 1–14 number of α-impacts on the paths are important for understanding the bugs. Our further investigation discovered that although in general, the faults were discovered by only propagating through several procedures, our detectors are able to identify faults deeply embedded in the program which cross the maximum of 7 procedures.

Table 2: Path Segment Computation

Benchmarks	Buffer		Integer		Pointer	
	α-s	α-i	α-s	α-i	α-s	α-i
wuftp-1	3.9/6	6.8	-	-	-	-
sendmail-6	-	-	2.0/2	2.5	-	-
sendmail-2	2.0/2	6.5	-	-	-	-
polymophy-0.4.0	1.6/2	2.9	2.5/3	4.3	-	-
gzip-1.2.4	4.4/7	14.2	2.0/2	1.4	-	-
ffmpeg-0.4.8	-	-	1.6/2	2.7	3.0/3	3
tightvnc-1.2.2	-	-	3.6/4	7.7	-	-
putty-0.56	-	-	1.3/2	1.6	-	-
apache-2.2.4	2.0/2	8.0	2.5/3	3.8	3.0/3	3

5.3 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 is the Dell Precision 490, one Intel Xeon 5140 2-core processor, 2.33 GHz, and 4 GB memory.

In Table 3, Column *Size* provides the size of the benchmarks in terms of lines of code. Scalability data are collected for all three detectors. Column q reports the total number of queries raised for each analysis. The results indicate that the number of queries raised for a benchmark is not always proportional to the size of the benchmarks and the code characteristics also matter. Column t gives the time used for detecting faults. All the benchmarks are able to finish within reasonable time. The maximum time of 182.6 minutes is reported by analyzing `apache-2.2.4` for integer faults. The time used for analysis is also 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. To show the advantage of demand-driven analysis compared to exhaustive analysis, we also list the percentage of procedures and blocks visited in the analysis for detecting each type of fault. See Columns $\%p$ and $\%b$. Benchmarks `wuftp-1`, `sendmail-2` and `sendmain-6` are manually constructed and only contain the code that relates to the bug we identify. Therefore, most of the code is visited during analysis. Most of the procedures in the benchmark contain pointer dereferences, and thus the portion of code visited for null-pointer dereference detection is higher than the analyses for integer fault and buffer overflow. The data in Table 3 supports our hypothesis that a demand-driven analysis only visits a portion of relevant code. All of our experiments are able to finish using memory under 4 GB.

5.4 Interactions of Analyses

In the last experiment, we integrate the specifications of buffer overflow, integer error and null-pointer dereference and generate one analysis to handle all three types of faults. The goal of this experiment is to explore the benefits of an integrated analysis. We find that in 6 out of 9 benchmarks, there exists queries of different types that are raised at the same α-point, which means we can combine the constraints for different properties in one query and determine whether an α-point contains multiple types of faults in one propagation. We also discover that 2 out of 9 programs can take advantage of caching between different types of queries. For example, a buffer overflow query is resolved using the cache computed by the integer query. In addition, we find that among the 7 programs that report integer faults, all of them have buffer overflow queries that reach the integer faults. That is, the integer faults potentially change the safety of those buffers.

6. RELATED WORK

Much research has been done for static fault detection due to its importance. Representative path-sensitive tools include ESP [5], ESPx [10], Saturn [20], ARCHER [21], MOPs [4], and Marple [14]. None of these fault detectors have shown the scalability and capability to detect both data and control centric faults, as done in this paper. The main difference of ESP and our work is that ESP applies a heuristic to select the information that is relevant to the faults, while driven by the demand, our analysis is able to determine the usefulness of the information based on the actual dependencies of variables, achieving more precision. Compared to our work, Saturn is more expensive in that it performs a backward slicing before analysis to select nodes that are relevant to the faults. Since our analysis runs backwards, the data and control dependencies can be determined as the analysis goes, not incuring extra cost. Besides the two closely related work, ESPx and ARCHER apply exhaustive analysis, both for detecting buffer overflow. ESPx uses manual annotation for scalability and ARCHER relies on a bottom up modular analysis and an optimized constraint solver. MOPS detects typestate violations, using model checking based techniques. Our work improves on Marple in that we address the challenges of scalability for a more variety types of faults, and unify the detections on a framework that can automatically generate analyses from specifications.

Path-insensitive fault detectors are less precise but more efficient. A commonly applied technique is type based, which detects faults by enforcing type safety on C. The representative tools are CQual [6], CCured [17], and Rich [3]. Another type of tool applies dataflow analysis to detect specified fault patterns. Examples include FindBugs [9], Splint [8] and Metal [11]. These three all developed specification or annotation techniques to support a range of faults, but none of the tools automatically generate individual fault detectors.

Our work benefits from program slicing [19] 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 resolved 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.

Table 3: Scalability

Benchmarks	Size (kloc)	Buffer				Integer				Pointer			
		q	t	$\%p$	$\%b$	q	t	$\%p$	$\%b$	q	t	$\%p$	$\%b$
wuftp-1	0.4	13	51.7 s	100	64	0	-	-	-	18	3.7 s	60	45
sendmail-6	0.4	1	122.8 s	100	78	6	67.4 s	100	53	12	12.3	100	75
sendmail-2	0.7	24	16.3 s	57	66	7	2.3 s	71	32	55	21.9 s	65	57
polymophy-0.4.0	1.7	15	3.1 s	9	36	3	3.7 s	27	7	9	35.6 s	73	36
gzip-1.2.4	8.2	39	471.8 s	38	49	62	141.3 s	43	38	86	196.9 s	62	63
ffmpeg-0.4.8	39.8	307	88.1 m	62	57	410	33.6 m	56	42	1976	60.2 m	97	76
tightvnc-1.2.2	78.9	21	54.9 m	53	54	1480	18.3 m	59	40	847	56.6 m	100	78
putty-0.56	112.4	228	46.8 m	48	41	152	135.2 m	50	45	750	52.1 m	85	69
apache-2.2.4	418.8	518	96.2 m	68	55	423	182.6 m	63	45	2734	168.3 m	92	67

Demand-driven techniques have shown scalability in various domains such as pointer analysis [13], dataflow analysis [7], infeasible path detection [2] and bug detection [14, 15]. The demand-driven analysis can be path-sensitive [14] or path-insensitive [7], forward [15] or backward [2]. Our work is the first that develops a comprehensive demand-driven framework for path-sensitive fault detection and demonstrates its effectiveness.

7. CONCLUSION

In this paper, we present a unifying framework, which includes a general, scalable analysis, a specification technique, and a generator for automatically generating desired fault detectors. The generated analyses are path-sensitive and interprocedural. They return a path segment where a fault occurs. Our experiments show that the framework can generate scalable analyses that identify the common faults of buffer overflow, integer fault and null-pointer dereference. 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 faults, and also exploration of the potential parallelism that exists for computing query resolutions.

8. REFERENCES

[1] R. Bodik, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[2] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1997.

[3] D. Brumley, T. cker Chiueh, R. Johnson, H. Lin, and D. Song. Rich: Automatically protecting against integer-based vulnerabilities. In *In Symposium on Network and Distributed Systems Security*, 2007.

[4] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.

[5] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[6] R. J. David and D. Wagner. Finding user/kernel pointer bugs with type inference. In *In Usenix Security Symposium*, 2004.

[7] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 1997.

[8] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, 1996.

[9] FindBugs. http://findbugs.sourceforge.net/.

[10] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *Proceeding of the 28th International Conference on Software Engineering*, 2006.

[11] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[12] Y. Hamadi. Disolver : A Distributed Constraint Solver. Technical Report MSR-TR-2003-91, Microsoft Research.

[13] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2001.

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

[15] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *Proceedings of 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003.

[16] Manviur Das. Talk at workshop on the evaluation of software defect detection tools. 2005.

[17] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, 2002.

[18] Phoenix. http://research.microsoft.com/phoenix/.

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

[20] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[21] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003.

[22] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004.