Segmented Symbolic Analysis

Wei Le
Rochester Institute of Technology
One Lomb Memorial Drive, Rochester, NY, USA
wei.le@rit.edu

***Abstract*—Symbolic analysis is indispensable for software tools that require program semantic information at compile time. However, determining symbolic values for program variables related to loops and library calls is challenging, as the computation and data related to loops can have statically unknown bounds, and the library sources are typically not available at compile time. In this paper, we propose *segmented symbolic analysis*, a hybrid technique that enables fully automatic symbolic analysis even for the traditionally challenging code of library calls and loops. The novelties of this work are threefold: 1) we flexibly weave symbolic and concrete executions on the selected parts of the program based on demand; 2) dynamic executions are performed on the unit tests constructed from the code segments to infer program semantics needed by static analysis; and 3) the dynamic information from multiple runs is aggregated via regression analysis. We developed the Helium framework, consisting of a static component that performs symbolic analysis and partitions a program, a dynamic analysis that synthesizes unit tests and automatically infers symbolic values for program variables, and a protocol that enables static and dynamic analyses to be run interactively and concurrently. Our experimental results show that by handling loops and library calls that a traditional symbolic analysis cannot process, segmented symbolic analysis detects 5 times more buffer overflows. The technique is scalable for real-world programs such as putty, tightvnc and snort.**

I. Introduction

Since its first application in debugging in 1969 [1], [2], symbolic analysis has demonstrated its break-through capabilities in bug finding and test input generation [3], [4], [5], [6], [7]. Compared to dynamic testing, symbolic analysis explores program behaviors along as many execution paths as possible and is potentially able to reason for paths which randomly generated input cannot reach. Besides its coverage, symbolic analysis requires no executables and can be applied early in software lifecycle while fixing a bug is cheaper. Compared to traditional static analysis such as dataflow analysis, symbolic analysis tracks actual symbolic values and relations of variables at program points along program paths and is able to provide more precise information. Recent research has shown that by running symbolic analysis in parallel on clusters, we can make it applicable for complex software systems [8]. With such potential for scalability, symbolic analysis will continue being a powerful tool for extracting program semantic information for important software engineering tools.

Despite great potential, automatically determining symbolic conditions for a program faces several challenges. To construct symbolic conditions at statements, we need to interpret the semantics of statements and resolve pointer aliasing. Sometimes, program behaviors are not solely determined by source code, and thus symbolic analysis purely using source information can fail in reasoning library calls or machine dependent ambiguities such as rounding or overflow. In addition, deriving symbolic conditions for paths containing loops is difficult, as it is impractical to enumerate paths for all the loop iterations.

Existing approaches for handling libraries and loops require either manual effort or loss of precision. Bush et al. and Chipounov et al. manually developed library models in Prefix [9] and Cloud9 [7] respectively. As each new application may include a different library, these models are not always able to be reused. Godefroid et al. applied concolic testing [3], [4], where a program runs with both concrete and symbolic inputs. When an unknown library call or loop occurs in symbolic analysis, concrete values are used for constructing symbolic conditions. This approach is not precise and only handles those library calls and loops that a test input can reach. Saxena et al. built template symbolic updates for loops based on code patterns [10]. Cadar et al. and Chipounov et al. analyzed loops through only one iteration [7], [11]. In both of these approaches, very limited loop behaviors are reasoned.

The goal of our work is to enable fully automatic symbolic analysis even for traditionally challenging code such as loops and library calls. Our insights are 1) code is not uniformly easy to analyze, and we thus should not apply a uniform strategy for analyzing an entire program as traditional symbolic analysis does; 2) the capabilities of symbolic analysis are limited especially when handling loops, library calls and pointers, and we should introduce dynamic analysis to supply information that a pure static symbolic analyzer is slow or unable to produce; and 3) in a program, some statements are more cohesive among each other than others, and we should leverage the structural and semantic relations between statements for partitioning the program and applying different analyses accordingly.

In this paper, we propose *segmented symbolic analysis*, a hybrid technique that partitions a program on demand and invokes concrete executions on selected code segments for generating symbolic substitution rules, also called *transfer functions*, needed by symbolic analysis. Compared to traditional hybrid techniques where static and dynamic analyses are run in different phases and for an entire program, our segmented symbolic analysis has the following novel goals for creating and using the dynamic information:

1) **Fully automatic**. Running an entire program requires software executables and could be slow. If a program is large, randomly generated test inputs are not sufficient for achieving code coverage desired by dynamic anal-

ysis. To address these challenges, we aim to construct unit tests and only perform dynamic analysis on the code segments of interest. Since these test programs contain a small number of branches, we can achieve desired code coverage with automatically generated test inputs.

2) **Aggregated information**: One concrete execution is often not able to provide representative values at program points. We thus need a technique to aggregate the dynamic information obtained from different runs. Previous research shows that programs mostly consist of linear operations [12], [13], and determining program properties often only requires linear constraints [6], [14]. Based on these observations, we assume that linear relations can characterize relevant behavior of small code segments. Therefore, *regression analysis* [15] is a natural fit to derive linear symbolic relations between program variables.
3) **On-demand and concurrent framework**: Static and dynamic analyses are concurrently applied on different parts of the program to ensure that the capabilities of the analyses match the code characteristics. The interactions between static and dynamic analyses are on demand, accomplished via a query based protocol.

We developed a framework and tool, called *Helium*. It runs a symbolic analysis and multiple dynamic analyses requested by the symbolic analysis. The symbolic analysis determines symbolic conditions in a demand-driven, path-sensitive fashion [6], [16]. When a *static unknown* such as a library call or a loop is encountered, the symbolic analysis suspends the propagation of the current symbolic condition and continues analyzing other paths. Meanwhile, Helium performs a structural and def-use analysis of the code that causes *static unknown* and identifies the code segment for constructing a unit test. The dynamic analysis automatically synthesizes the unit test and generates the test inputs. After running the tests, a regression based inference is performed on test inputs and outputs to derive transfer functions for the unknown code segment. Notified with the newly discovered information, the symbolic analysis then updates the blocked symbolic conditions and resumes the analysis of this path.

We implemented Helium to identify infeasible paths and buffer overflows. We found through experimentation that segmented symbolic analysis is able to reason library calls and loops required for determining bugs and infeasible paths. Compared to traditional symbolic analysis, our segmented symbolic analysis reports improved detection capabilities and reduced static unknowns. The techniques can be generally applied to real-world programs such as *putty* and *snort*. Unlike other hybrid symbolic analysis, by running tests on small code segments for coverage and using regression analysis for summarization, our dynamic inference did not report an under-approximation or over-approximation.

In summary, the research contributions of the work include:

- a framework that enables on demand, interactive static and dynamic analyses for handling loops and library calls;
- regression based dynamic inference that abstracts the local program behaviors; and
- experimental results to demonstrate the capabilities of segmented symbolic analysis.

The remainder of the paper is organized as follows. In Section II, we provide an overview of the Helium framework. In Sections III to V, we explain each component of the framework respectively. In Section VI, we present the experimental results, followed by the related work in Section VII and conclusions in Section VIII.

II. The Helium Framework

We first use an example to intuitively explain how segmented symbolic analysis works. We then describe the three internal components of the framework.

A. An Overview of Segmented Symbolic Analysis

In Figure 1, we display the control flow graph (CFG) for the simple example on the left. On the right, we compare three different symbolic analyses in detecting a buffer overflow located at node 10. The goal is to show that our segmented symbolic analysis is able to find bugs that traditional symbolic analysis cannot.

Under *Traditional SA* in Figure 1, we show how a traditional symbolic analysis attempts to find this buffer overflow. Here, we use demand-driven, symbolic analysis [6] to explain the process. In the first step, we raise a query [*size(filename)>len(filename)+1*], inquiring at node 10, whether the size of buffer *filename* would be larger than or equal to the length of the string in the buffer. To determine the resolution of this symbolic condition, we propagate the query backwards along path $\langle 10 - 1\rangle$. At node 9, the analysis determines that variable *t* is not relevant to the query; we then advance the query to node 8. At node 8, the analysis is blocked by the library call *_stat64i32* and reports an unknown. Without further knowledge, we do not know whether input *filename* has been changed in this call.

Under *Traditional SA with Library Models*, we show that the analysis is able to continue after the developer models the library call *_stat64i32*. However, the analysis still would be blocked by a string manipulation related loop located at $\langle 5 - 7\rangle$ because we do not know the iteration of the loop at compile time.

In segmented symbolic analysis, we partition a program when static analysis is blocked. In Figure 1, we identify two code segments: 1) node 8 contains a library call and node 1 initializes its parameter; and 2) nodes 5–7 contain a loop and the initialization for its local variable *i* is done at node 4. Based on the two code segments, we construct unit tests, one for reasoning about *_stat64i32* (Figure 2) and the other for reasoning about the loop (Figure 3). Each test program consists of four parts: initializing variables with test inputs, exercising segmented code, reporting test outputs and cleaning up. In Helium, we construct *_GenChars* to generate input strings and *_ReturnSpace* to clean up the global buffer *g_buf* for reuse.

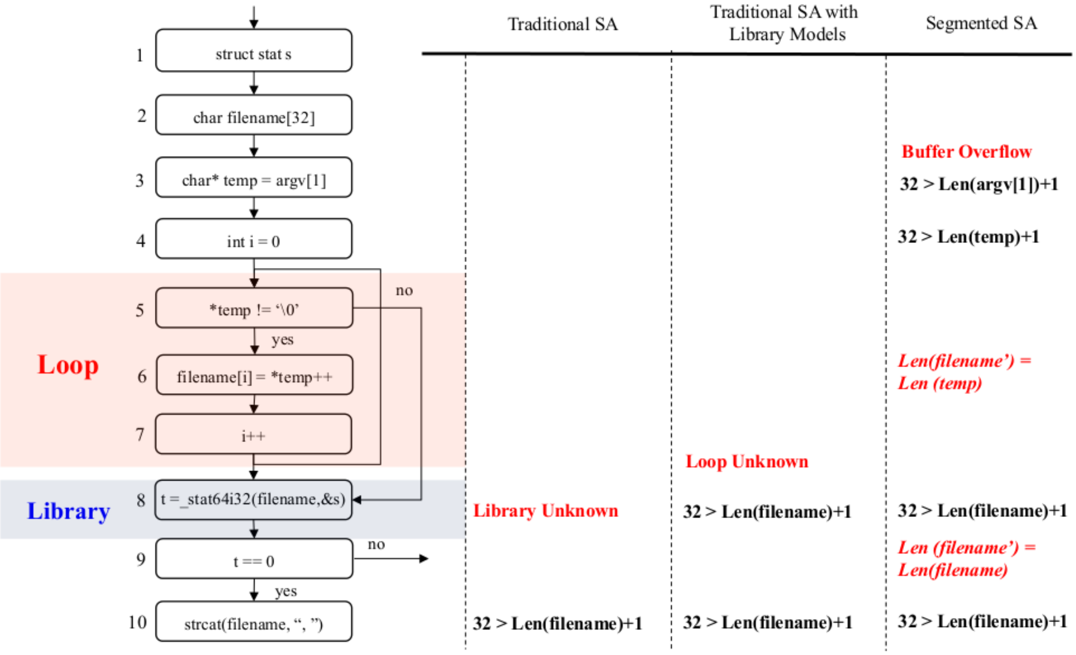

Fig. 1. An Example: Segmented Symbolic Analysis

```
//initialize with test inputs
char* filename = _GenChars(g_buf);

//code segment for the library call
struct stat s;
_stat64i32(filename, &s);

//output Len(filename)
char* _result = _GenChars(g_buf);
int _rint = strlen(filename);
itoa(_rint, _result, 10);
fputs(_result, fp);

//cleanup
_ReturnSpace(filename);
_ReturnSpace(_result);
```

Fig. 2. The test program for reasoning about *_stat64i32*

```
//initialize with test inputs
char* temp = _GenChars(g_buf);
char* filename = _GenChars(g_buf);

//code segment for the loop
int i = 0;
while(*temp != '\0'){
    filename[i] = *temp++;
    i++;
}
//output Len(filename) and cleanup
...
```

Fig. 3. The test program for reasoning about the loop

In Table I, we show 3 example sets of test inputs and outputs generated when testing the code in Figure 3. In this example, we generate inputs to initialize *temp* and *filename* before exercising the loop code. Both the string length and content are generated randomly, shown under *Test Input* in Table I. Since our regression analysis is performed on the integer domain, we predefined a set of mapping rules to map the test input to integers for performing regression analysis, shown under *Transformed Input for RA*. We select variables of interest, in this case, *filename*, for test output, shown under *Test Output*. Here, we use *filename'* to represent the value of *filename* after executing the code segment. Applying regression analysis to reason about the transfer function, we perform inference on the data under *Transformed Input for RA* and *Test Output*, using the linear model, $Len(filename') = \beta_0+\beta_1 Len(temp)+\beta_2 Len(filename)$. We obtain $Len(filename')= Len(temp)$. Similarly, we infer $Len(filename') = Len(filename)$ at the library call, indicating the length of the string *filename* would not be impacted by the library call. Performing symbolic substitutions using the discovered rules, we derive the query $[32 > Len(temp)+1]$ at node 4 and determine the buffer overflow at node 3. Path segment $\langle 3-10\rangle$ is reported as a buffer overflow.

TABLE I
TEST INPUT AND OUTPUT FOR REASONING ABOUT THE LOOP

Test	Test Input		Transformed Input for RA		Test Output
	temp	*filename*	Len (*temp*)	Len (*filename*)	Len(*filename'*)
1	*acde*	*piidaf*	4	6	4
2	*tazipda*	*qdd*	7	3	7
3	*ad*	*dafdalfll*	2	9	2

B. The Components

Helium takes program source as input and runs a fully automatic hybrid analysis, reporting either bugs or path feasibility. Shown in Figure 4, the framework consists of three components. The static component performs symbolic analysis

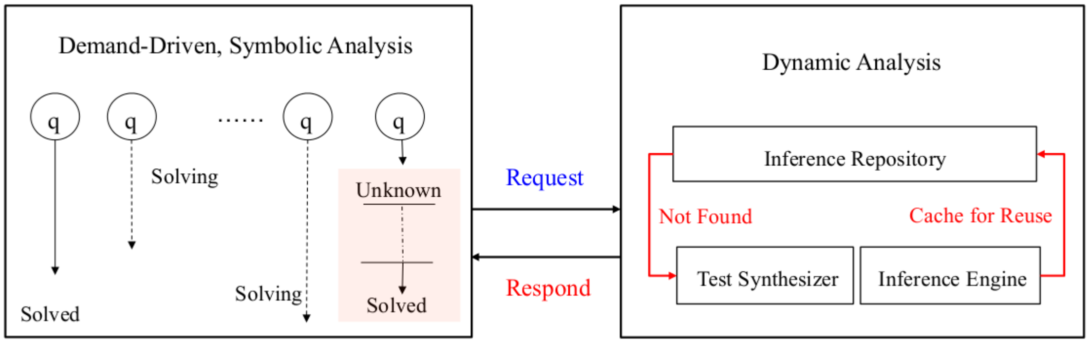

Fig. 4. The Helium Framework

and partitions programs. The symbolic analysis is demand-driven. It formulates a demand into queries at a set of program points of interest. For example, to detect infeasible paths, we raise queries at each conditional branch, inquiring whether the conditions at the branch can always be true or false [17]. A backward analysis is then performed to resolve the symbolic conditions in the queries [6]. When the analysis arrives at a statement where the rule for symbolic substitution has not been defined, the unknown code segment is partitioned from the program, and a request is formed to invoke dynamic analysis.

The second component is a communication protocol between static and dynamic analyses. The *request* enables transitions from static to dynamic analysis. It provides the dynamic analysis with: 1) the inquiries on the program variables, 2) the code segment for constructing the unit test, and 3) the constraints for generating test inputs. The *response* enables transitions from dynamic to static analysis. It contains program semantic information inferred by the dynamic analysis.

The dynamic analysis, shown on the right in Figure 4, consists of an *inference repository*, a *test synthesizer* and an *inference engine*. The test synthesizer takes the request from symbolic analysis and constructs a unit test for the code segment of interest. Based on the constraints provided in the request, the test synthesizer automatically generates inputs to run the program. The inference engine performs a regression analysis on the inputs and outputs of the tests and returns the discovered transfer functions and symbolic values. The repository stores all the inferred results for reuse.

III. Partitioning Programs On Demand

The challenges we addressed in the static component are to determine when and how to partition a program for constructing a meaningful test to reason static unknowns.

A. Demand-Driven Symbolic Analysis

Our explanations on symbolic analysis are based on the following language model. The language consists of:

1) data types: integer, Boolean, string, float/double, pointer and aggregate types composed of the above data types,
2) statements: assignments, arithmetic operations (+,-,*,/) on integer, float and double, *read()* (interface to get input data) and library calls, and
3) control structures: *if*, *then* and *else* statements, reducible loops and procedural calls.

Symbolic analysis, also called symbolic execution [18], (the term here refers to pure static symbolic analysis) takes symbolic inputs obtained at *read()* statements. During analysis, symbolic values of variables are collected at each statement. At the conditional branch *if* statement, symbolic analysis performs a fork, and symbolic conditions at *true* and *false* branches are collected respectively along each path. At the end of the paths, the symbolic conditions are sent to the constraint solver. The instances, if found, are the program inputs that can exercise the paths. If the error conditions are added, the instances are the inputs that can exercise the path and trigger the bug.

A typical symbolic analysis is exhaustive in that it visits the entire execution path, explores as many paths as the resource allows, and collects all symbolic values at the program points. In Helium, we develop a demand-driven symbolic analysis [6], [17], [16]. The symbolic conditions, e.g., regarding buffer safety or branch correlations, are only constructed at the selected program points of interest. The analysis starts from these conditions and performs a backward traversal of program paths to determine whether there exist inputs that can make the conditions to be *true* or *false*. Our analysis is *basic* in that it does not apply heuristics or manually constructed specifications. Its capabilities are summarized as follows.

Data types. We construct symbolic conditions for integers, Booleans, floats and doubles. We model string length and buffer size. For pointers and aggregate types, we apply an external, intraprocedural, field-sensitive pointer analysis [19]. We use *must-aliasing* and heuristically select *may-aliasing* [16] when constructing symbolic conditions.

Statements. We integrate the semantics of assignment (for all data types), arithmetic (on integer, Boolean, float and double) and the *read* function.

Control structures. We apply a demand-driven, interprocedural, path-sensitive, context-sensitive analysis [6], [20], [16] for propagating symbolic conditions across the control structures *if*, *then* and *else* and procedures. The symbolic analysis traverses each loop once for each path in the loop.

B. Partitioning Programs Based on Static Unknowns

Code partitioning is performed when the symbolic analysis encounters a statement or a control structure that it cannot analyze. In Helium, we consider two sources of static unknowns. We report an unknown at the library call if the variable under tracking is a global, an actual parameter of a pointer type, or the return of a library call. We report a loop unknown if we traverse a loop once and discover that the loop can have an impact on the symbolic conditions.

Since statically we cannot derive more information for such library calls and loops, we aim to run the code to infer their behaviors. Specifically, our goal is to determine the transfer functions, i.e., the relations of output and input variables for the unknown code segments. Ideally, we would run the entire program and perform dynamic inference based on the inputs and outputs observed at the entry and exit of the code segments. However, it is very hard to develop test inputs that can exercise any code segment of interest with high code coverage. Therefore, in Helium, we construct unit tests on the unknown code segments partitioned from the program. Although we do not run an entire program, we initialize program variables at the entry of unknown code using the test inputs that are equivalent or at least similar to the values we potentially obtain in a system testing. In this way, the unit tests simulate feasible program behaviors, and we can generate valid semantics for the unknown code.

To determine how to assign initial values for program variables in the test program, we first identify a set of variables used in the code segment but defined outside the segment. We perform a def-use analysis on the original program to decide where those variables are declared and defined. If a variable is initialized with some constant before entering the code segment, we include such initialization in the test program. For example, in Figure 3, we add *int i=0* when constructing a unit test for reasoning about the loop. In many cases, the variables used in the code segment are not constant. We apply a range analysis [19] to determine the potential ranges of these variables in the orignal program, based on which, we automatically generate test inputs to initialize the variables.

IV. Mixing Static and Dynamic Analyses

Different from techniques where static and dynamic components are run separately, our hybrid approach unifies the two types of analyses through online *requests* and *responses*. Besides efficiency, this approach makes a better use of dynamic analysis, as the dynamic analysis may not be able to reason symbolic values for every program variable but can answer a specific request from the static analysis.

A. From Static to Dynamic Analysis: Initializing Requests

Requests express the demand of static analysis and provide guidance for generating unit tests needed by dynamic analysis. Shown in Figure 5, a request is a 3-tuple $\langle V, C, E \rangle$, where:

1) V lists variables for which dynamic analysis should infer symbolic values. It defines the output variables for unit tests. The requests are expressed using *attributes*, e.g., *Value(v)* for the value of variable *v*, *Len(v)* for the length of string v and *Size(v)* for the size of buffer v.
2) C includes the code segment identified for constructing a unit test (see Section III-B).
3) E gives the types as well as the constraints of initial values, if any, for all the uninitialized variables used in the given code segment. It specifies the requirement for generating test inputs.

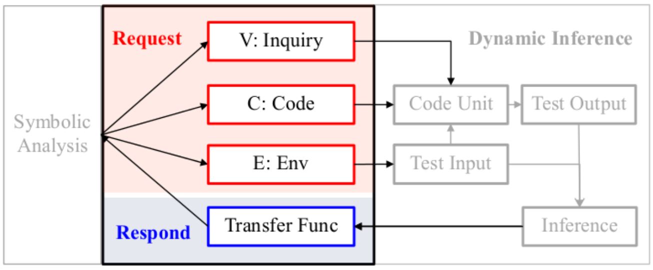

Fig. 5. Interaction Protocol: Request and Response

B. From Dynamic to Static Analysis: Consuming Responses

A *response* encapsulates any semantic information found by dynamic analysis on the code segment C related to V. In Helium, we currently support five types of models for representing the relevant behavior of a code segment, namely *constant*, *simple linear*, *multiple linear*, *polynomial linear* and *piecewise linear*, shown in Table II [15]. Dependent on whether the output variable a can be predicted using only one or multiple input variables, a linear relation can be simple or multiple. In a polynomial linear relation, the powers of input variables, e.g., b^2, or interactions of input variables, e.g. $c * d$, are used to represent the output variable a. There is also the piecewise linear model, where the linear relations can vary based on the ranges of the input variable(s). Given a request V from the static analysis, our dynamic inference component will automatically select a model that fits the data. The generated models will be supplied to static analysis for continuing symbolic analysis.

TABLE II
Explanatory Models for Representing Code Semantics
(Suppose a: Output Var, b, c, d: Input Vars)

Models	Examples
Constant	$a = 0$
Simple Linear	$a = b$
Multiple Linear	$a = 2 * b + c$
Polynomial Linear	$a = b^2 + c * d$
Piece-wise Linear	if $b > 0$ $a = b$, else $a = 3$

V. Inference On-Demand

In this section, we show how we construct a unit test from a given code segment and how we apply regression analysis to discover symbolic relations of program variables.

A. *Test Synthesizer*

A test synthesizer accomplishes two tasks. It constructs a test program from the code segment C with designated input and output variables, and it automatically generates a set of test inputs that satisfy E (see Section IV for definition).

Shown in Figure 6, we take four steps to construct a test program. First, we complete the code segment by including the function calls invoked and the libraries used. Second, we prune the code segments from which we are unlikely to infer precise transfer functions. In the current version of Helium, we use code size as heuristics to exclude such code segments. Specifically, we will exclude the code segments whose branch numbers or variable numbers exceed the thresholds. Our assumptions are that 1) if the code segment contains a small number of branches, we potentially can achieve a high test coverage even with randomly generated test inputs; 2) the transfer function for a smaller code segment is more likely to be linear; and 3) we are more likely to make a smaller code segment runnable. In our future work, we plan to iteratively decompose the code segment and perform segmented symbolic analysis within a segment until the confident linear relations are learned. In the final steps, we identify V in the code segment as test output variables and the variables whose initial values are not given as test input variables. We instrument the code for enabling such input and output in the test program.

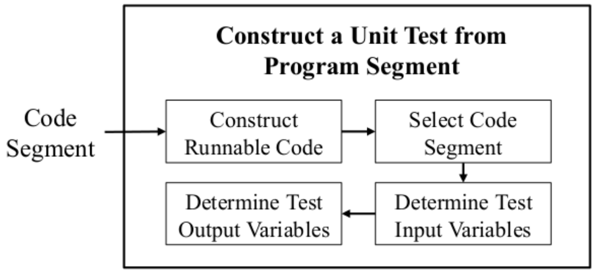
Construct a Unit Test from Program Segment
Code Segment
Construct Runnable Code
Select Code Segment
Determine Test Input Variables
Determine Test Output Variables

Fig. 6. Construct Unit Tests

In Helium, we support test input generation for both integer and string data types. Currently, we implement a random test input generator, taking the constraints identified for the test input variables into consideration. To determine the size of test inputs, there are two effective approaches. One approach is to stop generating and running test inputs when the desired code coverage is achieved for inferring confident symbolic relations. Another related approach is to stop testing when the transfer functions are learned and stabilized. Further research has to be done to determine such stopping criteria. In the current version of Helium, the test input size is not automatically determined, but configured by the users.

B. *Regression Based Inference*

After running the constructed tests, we obtain a set of values from test input and output. Based on these values, regression analysis is able to determine the relations between the input and output variables, if any.

1) Applying Regression Analysis to Discover Transfer Functions: In statistics, *linear regression* models the relationship between a scalar *response variable* y and one or more *explanatory variables* $X = \langle x_1, x_2, ..., x_n \rangle$ by fitting a linear equation to the observed data. The linear equation is called the *linear predictor function*, denoted

$$y = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n$$

where $\beta_0, \beta_1, ...$ and β_n are the coefficients, indicating the relative effect of a particular explanatory variable on the outcome [15].

Applying regression analysis to infer transfer functions, each program variable in request V is mapped to a dependent variable y and the input variables in the test program are mapped to the explanatory variables. For example, in Figure 1, to infer the impact of the library call for *filename*, we define the response variable *len(filename')* (the length of the string after the call) and the explanatory variable *len(filename)* (the length of the string before the call). The regression analysis makes inferences on the test inputs and outputs from multiple runs and determines *len(filename') = len(filename)*.

2) Steps of Regression Analysis: Since we consider all the program variables alive at the entry of unknown code as explanatory variables for regression analysis, we may end up including too many explanatory variables in a linear model and cause an over-fitting problem (the model describes a random error rather than the underlying relationships) [15]. To address this challenge, we apply *model selection* [15] to determine, for a specific response variable, which explanatory variables are valid. We then infer the relations for the response variable and the selected explanatory variables.

To reason polynomial linear relations that potentially exist among program variables, we perform a data transformation for the selected explanatory variables. For example, if we have two explanatory variables x_1 and x_2, we would construct the data $x_1 * x_2$, x_1^2 and x_2^2 for deriving potential polynomial relations $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 * x_2 + \beta_4 x_1^2 + \beta_5 x_2^2$. If regression analysis reports that $\beta_1 - -\beta_5$ are 0, the linear model is constant. Similarly, if $\beta_2 - -\beta_5$ are 0 or β_1 and $\beta_3 - -\beta_5$ are 0, the model is simple linear; if $\beta_3 - -\beta_5$ are 0, the relation is multiple linear. Otherwise, we obtain a polynomial model. If the above linear relations are not found, we resort to a piecewise regression model to fit the data.

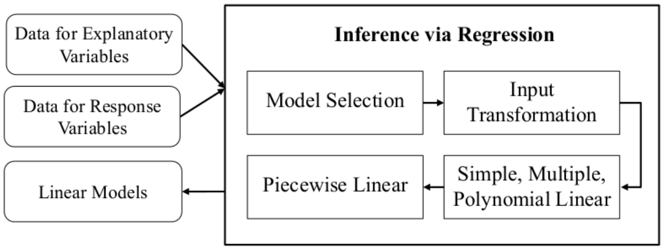
Data for Explanatory Variables
Data for Response Variables
Linear Models
Inference via Regression
Model Selection
Input Transformation
Piecewise Linear
Simple, Multiple, Polynomial Linear

Fig. 7. Inference via Regression

VI. Implementation and Experimental Results

The goals of our experiments are to determine 1) the capabilities of segmented symbolic analysis in handling loops and library calls and detecting bugs and infeasible paths; 2) the capabilities of regression based dynamic inference in discovering correct transfer functions; and 3) the scalability of concurrent, on-demand hybrid analysis.

A. Implementation and Experimental Setup

We developed the Helium framework for detecting infeasible paths and buffer overflows for C/C++ programs. We apply a branch correlation algorithm for infeasible path detection [17] and demand-driven analysis for buffer overflow detection [6]. The demand-driven, symbolic analysis is implemented using the Microsoft Phoenix compiler infrastructure [19] and the Disolver constraint solver [21]. Regression analysis is performed using the R statistical package [22]. Our experimental data are collected using the configurations of 100 test inputs per unit test and maximumly 10 input variables for constraining the test program size.

We collected 8 programs to evaluate our framework. The real-world programs such as *snort*, containing 99 k lines of code, are used to examine the scalability of the tool; *wu-ftpd* and *sendmail* from the buffer overflow benchmark [23], and *polymorph* and *gzip* from Bugbench [24], containing known buffer overflows, are used for estimating false negatives.

In our experiments, we ran a basic symbolic analysis (see Section III for description) as well as a segmented symbolic analysis for detecting both buffer overflows and infeasible paths. The comparison metrics include the number of bugs and infeasible paths detected, the number of static unknowns reported, and the time overhead. We also evaluated the capabilities of regression based dynamic inference by reporting the percentage of loops and library calls analyzed and the number of transfer functions generated.

B. Experimental Results

1) Comparing the Basic and Segmented Symbolic Analyses: In Table III, we compare the capabilities of the two symbolic analyses. On the left, we give the results for buffer overflow detection and on the right, we show the results for infeasible path detection. Under *SA* in the table, we list the data collected from the basic symbolic analysis and under *S-SA*, we display the results from the segmented symbolic analysis.

Under *PVS*, we show the total number of potential buffer overflow statements reported for each program. The demand-driven analysis raised queries from these statements. Comparing Columns *SA* and *S-SA* under *Overflow*, we find that the segmented symbolic analysis found a total of 32 buffer overflows for all the benchmark programs while the basic symbolic analysis reported only 4. We inspected the 32 buffer overflows and found that there are no false positives related to our dynamic analyses. There are 9 false positives among the 13 overflows reported for *snort*. Our inspection shows that these false positives are all related to the same infeasible path that we failed to detect due to the complex pointers. The basic symbolic analysis is blocked by a library call and does not encounter this infeasible path.

Among a total of 26 known buffer overflows located in *wu-ftpd*, *sendmail*, *polymorph* and *gzip*, we detected 18 and missed 8. Take *polymorph* as an example. We found a total of 7 of 8 known buffer overflows and we missed one overflow because our dynamic inference does not yet handle the library call *strrchr*. We have further discussions on the capabilities of segmented symbolic analysis in Section VI-C.

Under *Determined*, we list the number of statements that contain statically determined paths. Under *Unknown*, we show the number of statements that contain static unknown paths. For all 8 benchmarks, segmented symbolic analysis reports better detection capabilities and less unknowns.

To show that segmented symbolic analysis can be generally applied for determining different properties, we also compare the results for infeasible path detection. Under *Cond*, we list the number of conditional branches we analyzed for determining infeasible paths. Under *Infeasible*, we show that segmented symbolic analysis detects more infeasible paths than basic symbolic analysis for most benchmarks. For *sendmail*, pointer complexity blocks both of the symbolic analyses from further determining path feasibility, and thus the advantages of segmented symbolic analysis for handling loops and library calls are not applicable here. The results from this table also indicate that segmented symbolic analysis can handle both small and large programs. For the largest benchmark program, *snort*, we determined infeasible paths for 8 more conditional branches and reported 23 less unknowns than the basic symbolic analysis.

2) Feasibility and Capabilities of Dynamic Inference: Table IV summarizes the results from our dynamic inference, on the left for buffer overflow detection and on the right for infeasible path detection. Under *Segments*, we list the number of code segments we partitioned from a program. Under *Runnable*, we display the number of code segments that are successfully compiled and run. Under *Analyzable*, we give the number of unit tests, from which the semantic information is discovered. Finally, under *Inferred Items*, we list the total number of transfer functions discovered.

The results indicate that we inferred a total of 1135 models from all the benchmark programs. Take both buffer overflow and infeasible path detection into consideration. Segmented symbolic analysis successfully analyzed loops for 5 out of 8 programs and reasoned library calls for all the programs.

Comparing the data under *Runnable* and *Segments*, we show that we successfully generated unit tests for 29.3% of the loops encountered. Comparing *Segments* and *Analyzable*, we show that 23.8% of the encountered loops are successfully analyzed. We also managed to build unit tests for 81.1% and proccessed 70.4% of the library calls that block static analysis.

In our experiments, we found that symbolic equivalence—shown in the second row in Table II—is useful for modeling transfer functions for code segments that have no impact on the variables of interest. On the other hand, multiple and polynomial linear models shown in the third and fourth rows

TABLE III
SEGMENTED SYMBOLIC ANALYSIS FOR DETECTING BUFFER OVERFLOWS AND INFEASIBLE PATHS

Program	PVS	Overflow		Determined		Unknown	
		SA	S-SA	SA	S-SA	SA	S-SA
wu-ftpd	13	0	3	13	13	7	5
sendmail	24	0	3	21	21	18	16
polymorph-0.4.0	15	2	7	13	15	6	2
gzip-1.2.4	42	1	5	37	37	25	21
grep	8	1	1	8	8	6	6
tightvnc-1.2.2	27	0	0	21	22	12	11
putty-0.56	161	0	1	141	141	60	54
snort-1.9.1	147	0	13	110	121	53	45

Program	Cond	Infeasible		Determined		Unknown	
		SA	S-SA	SA	S-SA	SA	S-SA
wu-ftpd	8	1	2	2	3	4	3
sendmail	9	1	1	1	1	6	6
polymorph-0.4.0	7	3	4	3	4	5	4
gzip-1.2.4	35	9	11	11	13	24	22
grep	32	14	15	14	16	19	17
tightvnc-1.2.2	43	5	5	5	5	34	32
putty-0.56	199	30	31	30	31	72	70
snort-1.9.1	214	59	67	60	69	147	124

TABLE IV
DYNAMIC INFERENCE FOR BUFFER OVERFLOW AND INFEASIBLE PATHS DETECTION

Program	Segments		Runnable		Analyzable		Inferred Items
	loop	lib	loop	lib	loop	lib	
wu-ftpd	6	35	0	35	0	33	112
sendmail	7	26	7	25	7	16	79
polymorph-0.4.0	0	19	0	18	0	17	62
gzip-1.2.4	5	63	3	61	3	57	197
grep	2	7	0	5	0	5	11
tightvnc-1.2.2	8	11	0	5	0	2	4
putty-0.56	18	42	10	29	10	22	82
snort-1.9.1	37	47	11	40	9	25	148

Program	Segments		Runnable		Analyzable		Inferred Items
	loop	lib	loop	lib	loop	lib	
wu-ftpd	0	1	0	1	0	1	3
sendmail	0	5	0	5	0	5	17
polymorph-0.4.0	0	1	0	1	0	1	3
gzip-1.2.4	21	33	8	33	2	33	111
grep	6	8	0	4	0	4	12
tightvnc-1.2.2	9	16	0	0	0	0	0
putty-0.56	8	11	1	2	1	2	14
snort-1.9.1	26	104	3	84	3	79	280

in Table II are useful in modeling loop impact as well as the behaviors of some string library calls such as *strcat*. A piecewise linear model, shown in the fifth row in Table II, is a natural fit for reasoning the semantics of *if*, *then* and *else* statements or library calls such as *strncpy* and *strncat*.

Integrating dynamic information potentially leads to under-approximation and summarizing such information may lead to over-approximation. We manually inspected the transfer functions discovered by our dynamic inference and we did not find imprecise relations. This precision is achieved by running tests on small code segments for coverage, constraining test inputs based on the context of code segments in the orignial programs, and using regression analysis for summarization.

3) Scalability: Our experiments are run on a Windows machine with 4 duo Intel Core i7-2600 CPU and 16.0 GB memory. Each benchmark program is finished within 2.5 G memory. The time used for the two types of symbolic analyses are given in Table V.

Under *Size*, we display the sizes of the benchmark programs using lines of code. We show the time used for the basic symbolic analysis, infeasible path detection under *T-Inf* and buffer overflow detection under *T-Buf*. The results indicate that the time used for analysis is not linear to the program size. Since the analysis terminates when static unknowns are discovered, a larger program with more complex code may take a shorter time to analyze than a smaller program with simpler code.

We list the total time used for the segmented symbolic analysis under *Segmented Symbolic Analysis*/*T-Inf* and /*T-Buf*. We also give the total time used for the dynamic analyses, including constructing and running tests and performing regression analysis, under *I-Inf* and *I-Buf*. Since Helium concurrently runs symbolic and dynamic analyses, the accumulated time from multiple dynamic analyses could be more than the total time used by the segmented symbolic analysis. We display the number of threads invoked during the analysis under *Thread-Inf* and *Thread-Buf*.

Our experimental results show that segmented symbolic analysis generally takes longer to analyze a program compared to basic symbolic analysis, as it performs dynamic analyses. Also when a static unknown is unblocked, more code is potentially analyzed. Although slower, segmented symbolic analysis is still reasonably scalable. For the most expensive benchmark, *snort*, we finished infeasible path detection and buffer overflow detection both within an hour.

C. The Capabilities of Segmented Symbolic Analysis

In Table VI, we provide a summary of the types of loops and library calls Helium can and cannot handle. The first table on the left displays a set of libraries that we successfully analyzed. These are the library calls that manipulate strings, file systems and I/O. In the table labeled *LibNo*, we list types of library calls Helium currently does not handle. These library calls are the general challenges faced by any dynamic analysis. To infer the semantic information for calls such as *strrchr* and *getenv*, we need specially crafted test inputs to supply the string contents required to reason their behaviors. For calls such as *malloc*, we are not able to obtain the buffer size since the C language does not support a managed heap. For calls that access networking resources such as sockets, we need to model client-server behaviors in testing. For similar reasons, calls related to user interactions such as *getchar*, are not yet handled in our testing.

On the right in Table VI, we display a set of challenges for analyzing loops. The first row of the table indicates that we do not yet handle nested loops. Similar to library calls, we cannot handle loops that access network resources or involve

TABLE V
SCALABILITY OF SEGMENTED SYMBOLIC ANALYSIS

Benchmark	Size	Symbolic Analysis		Segmented Symbolic Analysis					
	(kloc)	T-Inf	T-Buf	T-Inf	I-Inf	Thread-Inf	T-Buf	I-Buf	Thread-Buf
wu-ftpd	0.4	0.7 s	1.5 s	2.7 s	4.0 s	6	523.8 s	580.0 s	206
sendmail	0.9	1.0 s	1.8 s	11.2 s	19.6 s	26	228.9 s	266.4 s	166
polymorph-0.4.0	0.9	2.2 s	1.0 s	3.9 s	3.4 s	6	143.3 s	171.6 s	96
gzip-1.2.4	5.1	358.7 s	3.0 s	1679.3 s	1683.9 s	271	508.6 s	598.0 s	341
grep	16.9	21.1 s	3.4 s	470.1 s	448.6 s	71	79.9 s	85.9 s	46
tightvnc-1.2.2	45.4	490.5 s	18.4 s	1149.9 s	850.0 s	126	596.6 s	425.5 s	96
putty-0.56	60.1	331.4 s	81.4 s	508.4 s	332.2 s	101	1213.6 s	1167.7 s	281
snort-1.9.1	98.8	124.6 s	465.6 s	2009.4 s	1893.4 s	651	1472.3 s	1121.1 s	421

TABLE VI
THE CAPABILITIES OF SEGMENTED SYMBOLIC ANALYSIS

LibYes: Type	Example
String	strcpy, strcat, strlen, strncpy, strdup
File System	chdir, getcwd, rename, unlink, stat
I/O	printf, fgets, fgetc, read
Misc	perror, utime, inet_addr,atoi

LibNo: Type	Example
String Content	strrchr, getenv
Compiler Unknown	malloc
Network Resource	recv, gethostbyname
Interactive Input	getchar

LoopNo: Type	Example
Complex Loops	nested loops
Network Resource	recv
Interactive Input	getchar
Invalid Context	Figure 8

user interactions. There are also loops that only can be run with a specially constructed test input. As an example, in Figure 8, we display a code snippet discovered in *tightvnc*. The code is used to render a window. The index of array *rcolors* in the loop is computed through a bit operation. Our randomly generated inputs are not always able to produce a valid index, and thus the unit test for this loop crashes from time to time.

```
//non−analyzable loop due to the invalid context
for (n = 7; n >= 8 − pfburh−>r.w % 8; n−−) {
  rcSource[i++] =
  rcolors[m_netbuf[y * bytesPerRow + x] >> n & 1];
}
```

Fig. 8. A Loop that Cannot be Handled by Segmented Symbolic Analysis

Segmented symbolic analysis is able to analyze loops that contain pointers, conditional branches and library calls, which static analysis is not capable to determine. In Figure 9 , we show an example discovered in the *sendmail* benchmark. This loop contains the library calls *isascii*, *isupper* and *tolower*, an *if* conditional branch and some pointer operations. Basic symbolic analysis only can determine that the string is potentially modified in the loop. Our dynamic analysis successfully reasons that the loop does not have an impact on *Len(p)*, and thus we can safely use the unchanged symbolic value for *Len(p)* at the loop exit.

```
//loop handled by segment symbolic analysis
for (p = name; *p != '\0'; p++){
  if (isascii((int)*p) && isupper((int)*p)){
    *p = tolower(*p);
    tryagain = TRUE;
  }
}
```

Fig. 9. The Power of Segmented Symbolic Analysis in Reasoning Loops

VII. RELATED WORK

Symbolic analysis has a variety of applications in software engineering. Examples include infeasible path detection [17], [6], test input generation [3], [4], bug detection [6], [11] and debugging [1]. Due to its importance, a set of symbolic analysis tools has been developed [3], [25], [4], [11], [7], [6]. Most of these tools are motivated by software testing with the goal of exploring as many paths as possible. For example, S2E [7] starts with a concrete execution and initiates symbolic execution at places where detailed checks are needed. Our work is motivated by precise static analysis in that we use dynamic analysis to generate a summary for code segments of interest, and our goal is to produce symbolic conditions for the code that static analysis is not able to automatically analyze.

In Table VII, we compare four representative hybrid symbolic tools. Under *Order*, we show that Check'n'Crash [26] and DSD [27] run static and dynamic analyses sequentially, while concolic testing [3], [4], [7] and segmented symbolic analysis run them concurrently. The difference between our tool and concolic testing is that segmented symbolic analysis can flexibly select code segments on demand for dynamic analysis, but concolic testing always runs tests for the entire program. Under *Size of Test Program*, we see that Check'n'Crash and DSD run tests for procedures and only report invariants at the entry or exit of a procedure. We also compare the four tools on where static and dynamic analysis exchange information, under *Interaction Location*, and what information to exchange, under *Exchange Information*. Compared to other tools, our analysis requires no executables or test inputs; the interactions of static and dynamic analyses are based on demand and thus more efficient; and the dynamic information is aggregated from multiple runs and thus is more confident.

Loops and library calls are challenges for all symbolic tools. Research has been done on refined local analysis to reason loops and library calls [28], [29], [30]. Sankaranarayanan et

TABLE VII
COMPARE FOUR TYPES OF HYBRID SYMBOLIC TOOLS

Hybrid Symbolic Analysis	Order	Size of Test Program	Interaction Location	Exchange Information
Check'n'Crash [26]	first static, then dynamic	procedure	likely buggy locations	safety constraints
DSD [27]	first dynamic, then static	procedure	entry of the procedure	program invariants
Concolic Testing [3], [4], [7]	concurrent, same code	entire program	static unknowns	some concrete value
Segmented Symbolic Analysis	concurrent, different code	code segment, on-demand	static unknowns	symbolic values, transfer functions

al. applied inductive logic programming and decision tree learning to infer specifications of libraries. The focus is to understand how the exceptions are handled in the library. Gopan et al. performed binary analysis to determine summaries for library calls. Our work provided linear models to approximate the behaviors of any small segments of code.

Our work is also related to discoveries of program invariants. Diakon instruments the programs, and based on the values at program points, it determines whether a predefined invariant potentially holds at a program point [31]. Compared to Diakon, we infer transfer functions rather than program invariants, and we apply regression analysis to determine the symbolic relations between variables.

VIII. Conclusions and Future Work

This paper presents segmented symbolic analysis, a novel, hybrid technique that flexibly weaves static and dynamic analyses on demand for their maximum capabilities of discovering program semantic information. The two key challenges we addressed here are: partitioning a program to construct valid unit tests for dynamic analysis, and mapping the problems of discovering symbolic relations between program variables to regression analysis. Our experimental results show that segmented symbolic analysis can address the loops and library calls that cannot be analyzed by traditional symbolic analysis. It is fully automatic and can be generally applied for determining different program properties and for different programs. Our future work includes extending the techniques to handle pointers and recursive calls and to further improve the code partition strategies.

Acknowledgments

We thank reviewers for their insightful comments.

References

[1] R. M. Balzer, "EXDAMS: expendable debugging and monitoring system," in *Proceedings of the May 14-16, 1969, spring joint computer conference - AFIPS '69 (Spring)*, May 1969, p. 567.

[2] L. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 215–222, Sep. 1976.

[3] K. Sen, D. Marinov, and G. Agha, "CUTE," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, p. 263, Sep. 2005.

[4] P. Godefroid, N. Klarlund, and K. Sen, "DART," *ACM SIGPLAN Notices*, vol. 40, no. 6, p. 213, Jun. 2005.

[5] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, no. 1, p. 20, Jan. 2012.

[6] W. Le and M. L. Soffa, "Marple: a demand-driven path-sensitive buffer overflow detector," in *Proceedings of the 16th International Symposium on Foundations of software engineering*, Nov. 2008, p. 272.

[7] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems," *ACM Transactions on Computer Systems*, vol. 30, no. 1, pp. 1–49, Feb. 2012.

[8] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 183–198.

[9] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software Practice and Experience*, 2000.

[10] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "loop-extended symbolic execution on binary programs," in *Proceedings of the 18th international symposium on Software testing and analysis*, 2009.

[11] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08, 2008.

[12] D. E. Knuth, "An empirical study of FORTRAN programs," *Software: Practice and Experience*, vol. 1, no. 2, pp. 105–133, Apr. 1971.

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

[14] N. Halbwachs, Y.-E. Proy, and P. Roumanoff, "Verification of Real-Time Systems using Linear Relation Analysis," *Formal Methods in System Design*, vol. 11, no. 2, pp. 157–185, Aug. 1997.

[15] M. Younger, *Handbook for Linear Regression*. Duxbury Press, 1979.

[16] W. Le and M. L. Soffa, "Marple: Detecting faults in path segments," *Transactions of Software Engineering and Methodology*, To Appear.

[17] 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.

[18] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[19] Phoenix, http://research.microsoft.com/phoenix/, 2005.

[20] W. Le and M. L. Soffa, "Generating analyses for detecting faults in path segments," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 320–330.

[21] Y. Hamadi, "Disolver: A Distributed Constraint Solver," Microsoft, Tech. Rep. MSR-TR-2003-91, 2007.

[22] "the r statistical package," http://www.r-project.org/, 2012.

[23] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *Proc. International Symposium on Foundations of Software Engineering*, 2004.

[24] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Proceedings of Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[25] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *Proceedings of the 2008 international symposium on Software testing and analysis*, ser. ISSTA '08, 2008.

[26] C. Csallner and Y. Smaragdakis, "Check 'n' crash: combining static checking and testing," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 422–431.

[27] ——, "DSD-Crasher: A hybrid analysis tool for bug finding," in *Proc. International Symposium on Software Testing and Analysis*, 2006.

[28] R. C. Waters, "A method for analyzing loop programs," *IEEE Trans. Softw. Eng.*, vol. 5, no. 3, pp. 237–247, May 1979.

[29] D. Gopan and T. Reps, "Low-level library analysis and summarization," in *Proceedings of the 19th international conference on Computer aided verification*. Berlin, Heidelberg: Springer-Verlag, 2007.

[30] S. Sankaranarayanan, F. Ivanči, and A. Gupta, "Mining library specifications using inductive logic programming," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 131–140.

[31] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *Proceedings of the 22nd international conference on Software engineering*, 2000, pp. 449–458.