Use Context-Free Grammar to Assist Symbolic Execution in Software Testing

Xusheng Xiao xxiao2@ncsu.edu
Xi Ge xge@ncsu.edu
Da Young Lee dlee10@ncsu.edu

Abstract

Symbolic execution is a way to track programs symbolicly rather than executing them with actual input value. With the impressive progress in constraint solvers, concolic path-based testing tools have literally blossomed up by combining both concrete and symbolic execution, which makes it possible to perform automatic path-based testing on large scale programs. However, these technologies are still suffering from the path explosion problem, since achieving 100% path coverage is to enumerate all paths between two nodes in a graph, which is well known as a NP-hard problem. To alleviate this path explosion problem and to reduce the computation complexity, different techniques has been proposed, such as pruning search space of symbolic execution, selective symbolic execution and fitness-guided path exploration. In this report, we provide the details on the problem of test data generation and present three techniques to alleviate this path explosion problem. We also discuss the early researchers who contributed to the field of research of test data generation using symbolic execution.

1 Introduction

To achieve automatic test (data) generation from source code, there are currently two well established frameworks: constraint-based testing [1, 2, 17, 31, 27] (CBT) and search-based testing [25, 26, 19, 23, 24] (SBT). CBT approach focuses on translating part of a program into a logical formula whose solutions are relevant to test data, while SBT approach is based on exploring the input space of the program using optimisation-like methods to guide the search toward relevant test data. In [19], Harman et al. show that SBT can be local or global. Similarly, CBT also can be global, translating the whole program in a formula [1, 2], or local (path-based) [17, 31, 27], focusing on a single path. In this report, we focus on path-based CBT, referred as path-based testing. Obviously such a local analysis is not sufficient to prove correctness of the whole system, however it is sufficient to derive a test data exercising the given path at runtime. Then, iterating the process on many different paths allows to automatically build test suites achieving a given structural coverage objective, e.g. branch coverage.

Symbolic execution [22] is a way to track programs symbolicly rather than executing them with actual input value. Concolic path-based testing tools have literally blossomed up recently [30, 4, 8, 9, 29, 33] with the impressive progress in constraint solvers. Concolic path-based testing tools combine both concrete and symbolic execution (referred as concolic execution [17, 31] or mixed execution [8]), which makes it possible to perform automatic path-based testing on large scale programs. By executing the program under test with concrete values while performing symbolic execution, symbolic constraints on the inputs can be collected from the predicates in branch statements, forming an expression, called path condition. To explore new paths, part of the constraints in the collected path conditions are negated to obtain new path conditions, which are sent to a constraint solver to compute test inputs for new paths. In theory, all feasible execution paths will be exercised eventually through the iterations of constraint collection and constraint solving in DSE.

A program under test can be modeled as a control flow graph (CFG) [3], whose nodes represent simple primitive statements (such as input, output, and assignment) and edges represent the flow of control. An execution path of the program is a path on CFG from the starting node, entry of the program, to the exit node, exit of the program. Thus, to explore all paths of the program under test, i.e. achieving 100% path coverage, is to enumerate all paths between two nodes in a graph, which is well known as a NP-hard problem [18]. To alleviate this path explosion problem and to reduce the computation complexity, different techniques has been proposed, such as pruning search space of symbolic execution [5], selective symbolic execution [10] and fitness-guided path exploration [35]. In this report, we provide the details on the problem of test data generation

and present three techniques to alleviate this path explosion problem. We also discuss the early researchers who contributed to the field of research of test data generation using symbolic execution.

2 HAMPI: A Solver for String Constraints

A lot of automatic analysis, testing, and verification tools can be reduced to a constraint generation phase and a constraint solving phase. The seperation of these two phases have leveraged more reliable and maintainable tools. In addition to that, increasing availability and efficiency of many off-the-shelf constraint solver makes the approach even more compelling. Hampi [21] is designed and implemented as a constraint solver for string-manipulating programs. Hampi constraints express membership in regualar language, fixed size context-free language and membership predicates. Given a set of constraints, hampi will give the string that satisfies all the constraints or report unsatisfiable. The experiment showes that Hampi is efficient in finding SQL injections by static and dynamic ananlysis on web applications and powerful in automated bug finding in system testing of c programs.

Many programs, like web applications, take string as inputs, manipulate them and then use them in sensitive operations as database queries. String constraint solver plays a very important role in automatic testing[6, 12, 28], verifying the correctness of program outputs[32], and finding security faults[13, 34]. Writing a string constraints solver is a very time-consuming work, and integrating it will cause less maintainable system. Therefore, Hampi is designed and implemented to meet this need as a third-party module that can be easily integrated into a variety of applications.

Hampi constraints express membership by regular language, fixed size context-free language. It may contain a fixed size string variable, context-free language definition, regular language definition and operations, and language-membership predicates. Given a set of string constraints over a string variable, Hampi outputs a string that satisfies all the constraints or reports that the constraints are unsatisfiable. Hampi is used as a component in testing, analysis, and verification applcations. Hampi can also be used to solve the intersection, containment, and equivalence problems for regular and fixed size context free languages.

A key feature for Hampi is that the fixed-sizing of regular and context free grammar. This feature differentiate Hampi with other string constraints solvers that used in many testing and analysis applications. Fixed-sizing is not a handicap for a constraint solver, but allows more expressive languages and many operations upon context-free language that would be undecidable without fixed-sizing. Fixed-sizing also renders the satisfiability problem solved by Hampi more tractable. Hampi works in four steps as indicated in figure 1: the first is to normalize the input constraints to formal forms which are

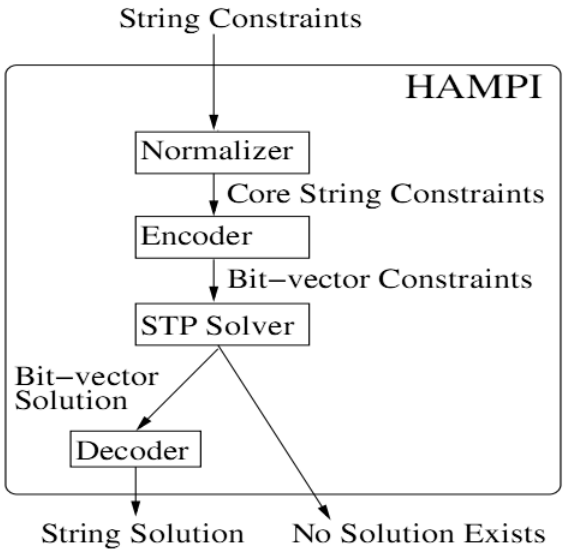

Figure 1. Schematic view of Hampi string solver.

called core string constraints. The core string constraints are expressions of the form $v \in R$ or $v \notin R$, where v is the input fixed-size string varible, and R is the regular expression. Second, translate the core string constraints into quantifier-free logic of bit-vectors which are fixed-size, ordered lists of bits. Third, hand over the bit-vector constraints logic that Hampi uses to

STP[14] which is a constraints solver for bit-vectors and arrays. Fourth, according to the report provided by STP, we get the result whether the original string constraints is satisfiable, if yes, generate a satisfying assignment in its bit-vector language and output a string solution; otherwise, report unsatisfiable. The procedure is illustrated as following graph:

We discuss the prominent feature and illustrate its language input by example. Hampi input enables the encoding of string constraint generated from the typical testing and security applications. The language supports the declaration of fixed-size variables and constraints, regular language operations, membership predicates, and the declaration of context free and regular languages, temporaries and constraints.

Input	::=	*Var Stmt**	HAMPI input
Stmt	::=	*Cfg* | *Reg* | *Val* | *Assert*	statement
Var	::=	**var** *Id* : *Int*	string variable
Cfg	::=	**cfg** *Id* := *CfgProdRHS*	context-free lang.
Reg	::=	**reg** *Id* := *RegElem*	regular-lang.
RegElem	::=	*StrConst*	constant
	|	*Id*	var. reference
	|	**fixsize**(*Id* , *Int*)	CFG fixed-sizing
	|	**or**(*RegElem* +)	union
	|	**concat**(*RegElem* +)	concatenation
	|	**star**(*RegElem*)	Kleene star
Val	::=	**val** *Id* :=*ValElem*	temp. variable
ValElem	::=	*Id* | *StrConst* | **concat**(*ValElem* +)	
Assert	::=	**assert** *Id* [**not**]? **in** *Id*	membership
	|	**assert** *Id* [**not**]? **contains** *StrConst*	substring

Figure 2. The Hampi Input Structure

Var is the string variable declared of the size specified. If all the constraints of the Hampi are satisfiable, var will be afforded value meets all the constraints. Sometimes, the application requires the constraint solver to consider all the string up to a fixed size. This end could be achieved by one of the following two ways: (1) repeatedly applying Hampi for different fixed size up to the given maximum size; (2) adjusting the constraints to allow "padding" of the variable.

Hampi allows the standard notation Extended Backus-Naur Form(EBNF) to specify context free grammar in input. Terminals are enclosed in double quotes(e.g., "SELECT"), and productions are seperated by vertical bar symble (|). Grammars may contain special symbols for repetition (+ and *) and character ranges(e.g.,[a-z]).

Reg is the declaration of regular language. Regular languages are defined as following four regular expressions:(i) a singleton set with a string constant; (ii) a concatenation or union of regular languages; (iii) a repetition of a regular language; (iv) a fixed sizing of a context free language. Every regular language can be defined by the first three of these operations.

Vals are temporary variables that act as shortcuts for expressing constraints on expressions that are concatenations of the string varibles and constants.

Assert is the key word that used by Hampi to express the membership of strings in regular languages.

After parsing all the Hampi input, Hampi normalize the string constraints into core form. The core string constraints are an internal intemediate representation that is easier to be encoded into bit-vector logic than raw Hampi input is. A core string constraints specifies membership in a regular language. A core string constraint is expressed in the form $StrExp \in RegExpor StrExp \notin RegExp$, where $StrExp$ is an expression composed of concatenations of string constants and occurrences of the string variable, and $RegExp$ is a regular expression.

The algorithm Hampi uses to create regular expressions that specify the set of strings of fixed length that are derivable from the context free grammar:

1. Expand all special symbols in the grammar, like repitition, option, character range.
2. Remove ϵ productions.
3. Take the following steps to construct the regular expression that encodes all fixed size strings of the grammar: (i) precompute the shortest and longest size of the string that can be generated from every nonterminal(i.e. upper bound and lower bound). (ii) given a size n and a nonterminal N, examine all the possible productions for N. For each $N \rightarrow S_1 S_2 ... S_k$, where each S_i could be nonterminal or terminal, enumerate all possible partitions of n characters to k grammar symbols. Then, create sub-expressions recursively and combine the sub-expressions together with a concatenation operator. Memoization the intermediate results makes this process scalable.

The next phase of Hampi is to encode the core string constraints as fomulas in the logic of fixed-size bit-vectors. A bit-vector is fixed size, ordered list of bits. The fragment of bit vector logic that is used by Hampi contains standard boolean operations,

extracting sub-vectors, and comparing bit vectors. Hampi asks STP for a satisfying assignment to the resulting bit-vector formula. If STP found one, Hampi decodes it and produce a string solution for the input constraints, otherwise Hampi will terminate and report that the string constraints is not satisfiable. The encode procedure is as follows:

1. The constant string values are enforced by Hampi as relevant elements of the bit-vector variable.
2. Hampi encodes the union operator(+) as a disjunction in the bit-vector logic.
3. Hampi encodes the concatenation operator by enumerating all possible distributions of the characters to the sub-expressions, encoding the sub-expression recursively, and combining the sub-formulas in a conjunction.
4. The Kleene Star will be encoded similarly to concatenation.
5. After STP finds a solution to the bit-vector formula (if exists), Hampi decodes the solution by reading 8-bit sub-vectors as consecutive ASC2 characters.

We will further illustrate the procedure by the following example:

Var v:2;
cfg $E :=$ "()"$|EE|$"("E")";
reg $Efixed := fixsize(E, 6)$;
Val $q := concat$("((", v, "))");
assert q in Efixed;
assert q contains "())";

Step 1: Normalize the constraints to core form. The results after this step are:
$c_1 : ((v)) \in ()[()() + (())] + [()() + (())]() + ([()() + (())])$
$c_2 : ((v)) \in [(+)] * ())[(+)]*$
Step 2: Encode the core form in bit-vector logic. We will illustrate how Hampi would encode constraint c_1. First of all, Hampi will create a bit-vector variable bv of size $48 = 6 * 8$ bits to represent the lefthand side of c_1. Second, the characters are translated into the ASC2 codes corresponding to them, "(" is 40, and ")" is 41. Then Hampi encode the lefthand side of c_1 as formula L_1, by specifying the constant value: L_1: $(bv[0] = 40) \wedge (bv[1] = 40) \wedge (bv[4] = 41) \wedge (bv[5] = 41)$. Byte $bv[2]$ and $bv[3]$ will be reserved for v, a 2-byte variable. Similarly, the right hand side of c_1 will be encoded as $D_{1a} \vee D_{1b} \vee D_{1c}$. The entire bit-vector logic of constraint c_1 after encoding would be $L_1 \vee (D_{1a} \wedge D1b \wedge D1c)$. The final formula that Hampi sends to STP solver is $(C_1 \wedge C_2)$.
Step 3: STP finds a solution that satisfies the formula: $bv[0] = 40, bv[1] = 40, bv[2] = 41, bv[3] = 40, bv[4] = 41, bv[5] = 41$. In the decoded ASC2, the solution is "(()())"(quote mark is not part of the solution string). Step 4: Hampi reads the assignment for variable v off of the STP solution, by decoding the elements of dv that corresponds to v, i.e., element 2 and 3. It reports the solutions for v as ")(".(even though there may be other solutions possible, STP can only find one.)

The performance of Hampi is evaluated through automatic finding of SQL injection attack strings by running a dynamic analysis tool on PHP web applications. Results show that Hampi has successfully replaced Ardilla[]'s custom attack generator, it solves the associated constraints quickly, finds all of the solution of $N \leq 6$, and solved all of the constraints in less than 10 seconds per constraint.

3 Grammar-based Whitebox Fuzzing Using Symbolic Execution

3.1 Whitebox Fuzzing

Whitebox fuzzing [28] executes the program under test with an initial, well-structured input, both concretely and symbolically. Along the execution, symbolic execution collects constraints on program inputs from the predicates in the conditional statements. The conjunction of these constraints of a execution path form an expression, called path condition. Satisfying the negation of each constraint in the path condition defines new inputs that exercise different control paths. Whitebox fuzzing repeats this process for the newly created inputs, with the goal of exercising many different control paths of the program under test and finding defects as fast as possible using various search heuristics. In practice, the search is usually incomplete because the number of feasible control paths grows exponentially with number of conditional statements in the program

under test and because the precision of symbolic execution, constraint generation and solving is inherently limited. However, whitebox fuzzing has been shown to be very effective in finding new security vulnerabilities in several applications.

In practice, the current effectiveness of whitebox fuzzing is limited when testing applications with highly structured inputs, e.g., compilers and interpreters. These applications process their inputs in stages, such as lexing, parsing and evaluation. Because of the enormous number of control paths in early processing stages, whitebox fuzzing rarely reaches parts of the application beyond these first stages. For instance, there are many possible sequences of blank-spaces/tabs/carriagereturns/etc. separating tokens in most structured languages, each corresponding to a different control path in the lexer. In addition to path explosion, symbolic execution may fail already in the first processing stages. For instance, lexers often detect language keywords by comparing their pre-computed, hard-coded hash values with the hash values of strings read from the input; this effectively prevents symbolic execution and constraint solving from ever generating input strings that match those keywords, since hash functions cannot be inversed (i.e., given a constraint x == hash(y) and a value for x, one cannot compute a value for y that satisfies this constraint).

In [15], Godefroid et al. propose a new approach, called *grammar-based whitebox fuzzing*, which enhances whitebox fuzzing with a grammar-based specification of valid inputs. They present a dynamic test generation algorithm where symbolic execution directly generates grammar-based constraints whose satisfiability is checked using a custom grammar-based constraint solver. Their algorithm consists of two key components:

1. Generation of higher-level symbolic constraints, expressed in terms of symbolic grammar tokens returned by the lexer, instead of the traditional [17, 9, 28] symbolic bytes read as input.

2. A custom constraint solver that solves constraints on symbolic grammar tokens. The solver looks for solutions that satisfy the constraints and are accepted by a given (context-free) grammar.

3.2 Example

Their algorithm never generates non-parsable inputs, i.e., the inputs generates can be accepted by the lexer and parser. In addition, the grammar-based constraint solver can complete a partial set of token constraints into a fully-defined valid input, thus avoiding exploring many possible non-parsable paths. By restricting the search space to parsable inputs, grammar-based whitebox fuzzing can explore deeper paths, and focus the search on the harder-to-test, deeper processing stages.

```
1 // Reads and returns next token from file.
2 // Terminates on erroneous inputs.
3 Token nextToken(){
4   ...
5   readInputByte();
6   ...
7 }
8
9 // Parses the input file, returns parse tree.
10 // Terminates on erroneous inputs.
11 ParseTree parse(){
12   ...
13   Token t = nextToken();
14   ...
15 }
16
17 void main(){
18   ...
19   ParseTree t = parse();
20   ...
21   Bytecode code = codeGen(t);
22   ...
23 }
```

```
FunDecl   ::=  function id ( Formals ) FunBody
FunBody   ::=  { SrcElems }
SrcElems  ::=  ϵ
SrcElems  ::=  SrcElem SrcElems
Formals   ::=  id
Formals   ::=  id , Formals
SrcElem   ::=  ...
...
```

Figure 3. Interpreter and fragment of a context-free grammar for JavaScript.

Consider the interpreter sketched in Figure 3 and the JavaScript grammar partially defined in Figure 3. By tracking the tokens returned by the lexer, i.e., the function nextToken in Figure 3, and considering those as symbolic inputs, their dynamic test generation algorithm generates constraints in terms of such tokens. For instance, running the interpreter on the valid input:

$$function\, f() \{\}$$

This input may correspond to the sequence of symbolic token constraints:

$$\begin{gathered} token_0 = function \\ token_1 = id \\ token_2 = (\\ token_3 =) \\ token_4 = \{ \\ token_5 = \} \end{gathered}$$

Negating the fourth constraint in this path constraint leads to the new sequence of constraints:

$$\begin{gathered} token_0 = function \\ token_1 = id \\ token_2 = (\\ token_3 \neq) \end{gathered}$$

There are many ways to satisfy this constraints but most solutions lead to non-parsable inputs. In contrast, our grammar-based constraint solver can directly conclude that the only way to satisfy this constraint while generating a valid input according to the grammar is to set:

$$token_3 = id$$

and to complete the remainder of the input with, say,

$$\begin{gathered} token_4 =) \\ token_5 = \{ \\ token_6 = \} \end{gathered}$$

Thus, the generated input that corresponds to this solution is:

$$function\ f(id)\ \{\}$$

Here id can be any identifier. Similarly, the grammar-based constraint solver can immediately prove that negating the third constraint in the previous path constraint, thus leading to the new path constraint:

$$\begin{gathered} token_0 = function \\ token_1 = id \\ token_2 \neq (\end{gathered}$$

is unsolvable, i.e., there are no inputs that satisfy this constraint and are recognized by the grammar. Grammar-based whitebox fuzzing prunes in one iteration the entire subtree of lexer executions corresponding to all possible nonparsable inputs matching this case.

3.3 Grammar-based Extension to Whitebox Fuzzing

Grammar-based whitebox fuzzing is an extension of the algorithm of whitebox fuzzing [28]. Their algorithm requires a grammar G that describes valid inputs. Instead of marking the bytes in program inputs as symbolic, grammar-based whitebox fuzzing marks tokens returned from a tokenization function, such as nextToken in Figure 3 as symbolic; thus grammar-based whitebox fuzzing associates a symbolic variable with each token, and symbolic execution tracks the influence of the tokens on the control path taken by the program P. The algorithm uses the grammar G to require that the new input not only satisfies the alternative path constraint but is also in the language accepted by the grammar. As the examples in the introduction illustrate, this additional requirement gives two advantages to grammar-based whitebox fuzzing: it allows pruning of the search tree corresponding to invalid inputs (i.e., inputs that are not accepted by the grammar), and it allows the direct completion of satisfiable token constraints into valid inputs.

3.4 Context-free Constraint Solver

A context-free constraint solver takes as inputs a context-free grammar G and a regular expression R, and returns either a string $s \in L(G) \bigcap L(R)$, or $\perp$ if the intersection is empty. They gives an algorithm to describe an decision procedure for such a constraint solver. They also provide three technical steps to eliminate recursion for the start symbol S: (1) Assign G to G'; (2) Duplicate productions for starting nonterminal S by creating S' in G'; (3) rename S to S' (but not in the duplicated productions). The algorithm exploits the fact that, by construction, any regular language R always constrains only the first n tokens returned by the tokenization function, where n is the highest index i of a token variable $token_i$ appearing in the constraint represented by R. The algorithm starts by converting the path constraint into a regular expression R. This is straightforward and involves grouping the constraints in pc by the token variable index. The algorithm employs a simple unroll-and-prune approach: in the ith iteration of the main loop, the algorithm unrolls the right-hand sides of productions to expose a $0 \ldots i$ prefix of terminals, and prunes those productions that violate the constraint c_i on the ith token variable $token_i$ in the regular expression R. During each round of unrolling and pruning, the algorithm uses the worklist W to store productions that have not yet been unrolled and examined for conformance with the regular expression.

After the unrolling and pruning, the algorithm checks emptiness [20] of the resulting language $L(G')$ and generates a string s from the intersection grammar G'. For speed, their implementation uses a bottom-up strategy that generates a string with the lowest derivation tree for each nonterminal in the grammar, by combining the strings from the right-hand sides of productions for nonterminals. This strategy is fast due to memorizing strings during generation.

To illustrate the algorithm, they provides an example, a simplified S-expression grammar. Starting with the initial grammar, the algorithm unrolls and prunes productions given a regular path constraint. The grammar is (S is the start symbol, nonterminals are uppercase).

$$S := (let((id\ S))\ S)|(Op\ S\ S)|num|id$$
$$Op := +|$$

and the regular path constraint is:

$$token_1 = \{(\}$$
$$token_2 = \{+\}$$
$$token_3 = \{(\}$$
$$token_4 = \{(,), num, id, let\}$$

Before the main iteration, the grammar is:

$$S' := (let((id\ S'))\ S')|(Op\ S'\ S')|num|id$$
$$Op := +|$$
$$S := (let((id\ S'))\ S')|(Op\ S'\ S')|num|id$$

Next, the main iteration begins. The first conjunct in the grammar constraint is $token_1 \in \{(\}$, therefore the algorithm removes the last two productions from the grammar. The result is the following grammar.

$$S' := (let((id\ S'))\ S')|(Op\ S'\ S')|num|id$$
$$Op := +|$$
$$S := (let((id\ S'))\ S')|(Op\ S'\ S')$$

In the next iteration of the for loop, the algorithm examines the second conjunct in the regular path constraint, $token_2 \in \{+\}$. The algorithm prunes the first production rule from S since let does not match $+$, and then expands the nonterminal Op in the production $S := (Op\ S'\ S')$. The production is replaced by two productions, $S := (+\ S'\ S')$ and $S := (-\ S'\ S')$, which are added to the worklist W. The grammar G' is then

$$S' := (let((id\ S'))\ S')|(Op\ S'\ S')|num|id$$
$$Op := +|$$
$$S := (+\ S'\ S')|(-\ S'\ S')$$

In the next iteration, the second of the new productions is removed from the grammar because it violates the grammar constraint. After the removal, the execution is now again at the top of the iteration.

$$S' := (let((id\ S'))\ S')|(Op\ S'\ S')|num|id$$
$$Op := +|$$
$$S := (+\ S'\ S')$$

After 2 more iterations of the for loop, the algorithm arrives at the final grammar:

$$S' := (let((id\ S'))\ S')|(Op\ S'\ S')|num|id$$
$$Op := +|$$
$$S := (+\ (let\ ((id\ S'))\ S')\ S')$$

As the last two steps, the algorithm checks that $L(G') \neq \emptyset$, and generates a string s from the final grammar G' for the intersection of G and R. Our bottom-up strategy generates the string $S := (+\ (let\ ((id\ num))\ num)\ num)$. From this string of tokens, our tool generates a matching string of input bytes by applying an application-specific detokenization function.

4 symbolicgrammar

5 Major Contributors

The concept of symbolic execution was introduced academically with descriptions of the Select system, proposed by Boyer et al [7]. In 1976, test data generation using symbolic execution was first proposed by James C. King [22]. Around the same time, Clarke also did the work on this [11]. Their pioneer works open the ways to automatic test data generation by using symbolic execution to do static analysis of code for path safety and prove theorems about code. However, these static ways faced the exponential state space explosion problem, which made it only practical for small programs. In recent years, Koushik Sen, whose paper on concolic testing [17] won the ACM SIGSOFT Distinguished Paper Award at ESEC/FSE '05, proposed CUTE and DART tools, blossoming up the path-based automatic test data generation using symbolic execution. With the impressive progress of constraint solvers and concolic path-based testing [30, 4, 8, 9, 29, 33], it is possible to perform automatic path-based testing on large scale programs. Pex [33], a symbolic execution test generation tool for .NET proposed by Nikolai Tillmann, has been used to test .NET core libraries and found serious bugs. To alleviate the classic path explosion problem, many new techniques are also been proposed by Xie [35], Godefroid [16] and so on.

References

[1] B. Botella A. Gotlieb and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *ISSTA*, 1998.

[2] B. Botella A. Gotlieb and M. Watel. Inka: Ten years after the first ideas. In *ICSSEA*, 2006.

[3] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, Cambridge, UK, 2008.

[4] S. Bardin and P. Herrmann. Structural Testing of Executables. In *ICST*, 2008.

[5] Sébastien Bardin and Philippe Herrmann. Pruning the Search Space in Path-Based Test Generation. In *ICST*, 2009.

[6] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 307–321, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT–a formal system for testing and debugging programs by symbolic execution. In *ICRS*, 1975.

[8] Cristian Cadar and Dawson Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *SPIN*, 2005.

[9] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS*, 2006.

[10] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective Symbolic Execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[11] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *TSE*, 1976.

[12] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 151–162, New York, NY, USA, 2007. ACM.

[13] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting sql injection vulnerabilities. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 87–96, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Vijay Ganesh. *Decision procedures for bit-vectors, arrays and integers*. PhD thesis, Stanford, CA, USA, 2007. Adviser-Dill, David L.

[15] P. Godefroid, A. Kiezun, and M. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, pages 206–215, 2008.

[16] Patrice Godefroid. Compositional Dynamic Test Generation. In *POPL*, 2007.

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[18] Jonathan Gross and Jay Yellen. *Graph theory and its applications*. CRC Press, Inc., Boca Raton, FL, USA, 1999.

[19] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ISSTA*, 2007.

[20] J. Hopcroft and J. Ullman. *Introduction to automata theory languages and computation.* Addison-Wesley Series in Computer Science, 1979.

[21] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: a solver for string constraints. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116, New York, NY, USA, 2009. ACM.

[22] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[23] B. Korel. Automated Software Test Data Generation. In *TSE*, 1990.

[24] B. Korel. Automated Test Data Generation for Programs with Procedures. In *ISSTA*, 1996.

[25] A. P. Mathur N. Gupta and M. L. Soffa. Automated Test Data Generation Using an Iterative Relaxation Method. In *FSE*, 1998.

[26] A. P. Mathur andM. L. Soffa. N. Gupta. UNA Based Iterative Test Data Generation and its Evaluation. In *ASE*, 1999.

[27] B. Marre N. Williams and P. Mouy. On-the-Fly Generation of K-Path Tests for C Functions. In *ASE*, 2004.

[28] M. Levin P. Godefroid and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.

[29] M. Y. Levin P. Godefroid and D.Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.

[30] C. S. Pasareanu S. Anand and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *TACAS*, 2007.

[31] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE*, 2005.

[32] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 13–22, Washington, DC, USA, 2007. IEEE Computer Society.

[33] N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In *TAP*, 2008.

[34] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260, New York, NY, USA, 2008. ACM.

[35] Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, June-July 2009.