ARTICLE IN PRESS

Future Generation Computer Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

State of the art: Dynamic symbolic execution for automated test generation

Ting Chen [a,*], Xiao-song Zhang [a], Shi-ze Guo [b], Hong-yuan Li [a], Yue Wu [a]

[a] School of Computer Science & Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China

[b] The Institute of North Electronic Equipment, Beijing 100191, China

ARTICLE INFO

Article history:
Received 17 June 2011
Received in revised form
21 December 2011
Accepted 20 February 2012
Available online xxxx

Keywords:
Dynamic symbolic execution
Automated test generation
Low false positives
High code-coverage
Challenges and solutions
Tools analysis
Prospects

ABSTRACT

Dynamic symbolic execution for automated test generation consists of instrumenting and running a program while collecting path constraint on inputs from predicates encountered in branch instructions, and of deriving new inputs from a previous path constraint by an SMT (Satisfiability Modulo Theories) solver in order to steer next executions toward new program paths. It has been introduced into several applications, such as automated test generation, automated filter generation and malware analysis mainly for its two intrinsic properties: low false positives and high code-coverage. In this paper, we focus on the topics that are closely related to automated test generation. Our contributions are five-fold. First, we summarize the theoretical foundation of dynamic symbolic execution. Second, we highlight the challenges when turning ideas into reality. Besides, we describe the state-of-the-art solutions including advantages and disadvantages for those challenges. In addition, twelve typical tools are analyzed and many properties of those tools are censused. Finally, we outline the prospects of this research field in detail.

1. Introduction

1.1. Background

Dynamic Symbolic Execution (DSE) for automated test generation consists of instrumenting and running a program while collecting path constraint on inputs from predicates encountered in branch instructions, and of deriving new inputs from the previous path constraint by an SMT (Satisfiability Modulo Theories) solver in order to steer next executions toward new program paths. DSE has become an attractive technique in the software engineering research for its two intrinsic properties. First, DSE does not suffer from significant false positives because the executing program affords enough runtime information. Typical DSE tools such as Fuzzgrind [1] and SAGE [2] produce alarms only when software exceptions are occurred in runtime. Second, in general, each new input generated by a solver is able to exercise new path, thus DSE leads to high code-coverage by traversing all program paths.

Symbolic execution has been studied since 70's [3,4]. But in the past 30 years, effective solutions and tools have proven elusive for several insurmountable technical challenges at that time. In recent years, thanks to powerful modern computers, potent symbolic execution engines, excellent constraint solvers, practical software model checkers, efficient theorem provers and precise yet scalable static analysis tools, symbolic execution has developed a lot. Now DSE becomes one of the hottest spots in the software engineering research. For instance, three papers in 2010 IEEE Symposium on Security and Privacy focused on this technique [5–7]. Quite a few applications utilize DSE and here are some examples.

Automated test generation. DSE is introduced in automated test generation for its low false positives and high code-coverage. This technique is now the foundation of several vulnerability discovery tools [2,8–12]. The majority of these tools are capable of automatically finding well-defined bugs, such as buffer overflows, divisions by zero, NULL pointer dereferences, etc.

Automated filter generation. Intrusion Detection/Prevention System (IDS/IPS) and anti-virus software always take advantage of input filters to block inputs that trigger bugs and vulnerabilities. A few years ago, input filters could only be produced by manual work. In order to contain fast spreading malcode, several automated input filters generation tools which aim at finding common byte sequences in different copies of one malcode are proposed [13–15]. Recently DSE has been absorbed into this research branch so as to find vulnerability signatures—a vulnerability signature is used to detect/prevent all kinds of attacks that target the specified vulnerability [16–20].

Malware analysis. Automated reverse-engineering techniques for malware analysis have employed DSE these years. Hence, the ability of analyzing how information flows through a malware

* Corresponding author.
E-mail addresses: chenting19870201@163.com (T. Chen), johnsonzxs@uestc.edu.cn (S.-z. Guo).

0167-739X/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2012.02.006

Table 1
Challenges and solutions of DSE.

Challenges	Solutions
Path explosion [37]	Intelligent search strategies [2,8,38]
	Summary [39–42]
	Symbolic grammar [43,44]
	Length abstraction [45]
	Information partition [46]
Extern/complex arithmetic functions	Concretization
	Function summary [39–41]
Floating-point computations	Concretization
	Static analysis [47]
Symbolic pointer operations	Concretization
	Novel memory model [48]
Environmental interaction	SC-UE model [10–12]
	SC-SE model [8]
	Symbolic tracking SQL queries [49]
	Mock databases [50]
	Collaborative input space exploring [6]

binary and exploring behaviors to trigger malware is remarkably raised [21–24].

In this paper, we only focus on the topics that are closely related to DSE based automated test generation. We feel necessary to clarify that two research fields—model checking based and search based test generation which seem to be close to DSE are out of the range of this article. Model checking [25,26] is a widely-accepted verification technique which takes a temporal logic property and analyzes the model in order to determine whether the model violates the property or not. A nature feature of model checking is the ability to generate witnesses and counter-examples for property satisfaction or violation respectively [27,28]. From software testing aspect, the counter-examples can be interpreted as test cases. DSE and model checking based test generation vary a lot in terms of methodology by their definitions even though they have similar purpose. Actually, these two research branches overlap to some extent and their collaboration will be described in more detail in Section 7.

Test generation could be considered as an optimization question that finding solutions to the test criteria via search based algorithms [29,30]. There is a significant discrepancy between DSE and search based test generation that the former uses an SMT solver to obtain solutions directly while the latter utilizes search algorithms such as hill climbing [31], simulated annealing [32–34], genetic algorithms [35,36] etc. to approach solutions. In Section 7, we will see that those two techniques could be integrated in order to cope with more complicate programs and get higher code coverage with fewer test cases.

1.2. Contributions

Our contributions are five-fold. First, we summarize the theoretic foundation of DSE, including formal representation and conceptual descriptions. Second, we underline the challenges when turning ideas into reality. In addition to above two, we describe the up-to-date solutions. We have to note here that existing solutions can only partially address those challenges and the ideal alternatives are still in the research. Here, we briefly list all challenges and related solutions referred in this paper in Table 1.

The fourth contribution is that we compare and analyze twelve typical DSE tools in Section 6. Many features such as platform, instrumentation tools, SMT solvers etc. are compared in detail. Researchers and practitioners will benefit from this section to choose proper DSE tools for their own purposes or make decision to develop new DSE tools if current ones cannot meet their requirements. Moreover, we discuss the prospects of DSE for automated test generation in detail. Here, we just briefly list the trend: new techniques to figure out path explosion, more powerful SMT solvers, particular security tools for particular applications, manageable summary database and parallel DSE.

1.3. Paper layout

The remainder of this paper is organized as follows. Section 2 summarizes the theoretic foundation of DSE and then we present a simple example to illustrate how to generate tests automatically by means of DSE. Section 3 explains why we need DSE, i.e., we present the drawbacks of static techniques and black-box fuzzing as well as the advantages of DSE. Section 4 focuses on the challenges when turning ideas into reality and then we introduce several state-of-the-art solutions against those challenges. Section 5 describes some popular techniques that aim at improving efficiency of DSE. Section 6 compares and analyzes twelve typical DSE tools from many aspects. Section 7 mainly discusses the prospects of DSE for automated test generation in detail. Section 8 concludes this paper.

2. Definition of DSE

In this section, we first summarize the theoretic foundation of DSE, including formal representation and conceptual descriptions. Then we give a simple example to illustrate how to generate tests automatically by means of DSE.

We present the widely accepted definition by quite a few subsequent studies [24,39,51] of DSE which is first proposed in DART [11]. Equivalent definition in an abstract imperative programming language can be found in [45,46,52].

DSE consists of running the program P under test both concretely, executing the actual program, and symbolically, gathering constraints on values stored in program variables v and expressed in terms of input parameters. Side-by-side concrete and symbolic executions are performed using a concrete store M and a symbolic store S, which are mappings from memory address (where program variables are stored) to concrete and symbolic values respectively. These side-by-side executions require the program P to be instrumented at the level of a RAM (Random Access Memory) machine. Fortunately, a number of RAM-machine-level instrumentation tools are available. For example, CIL (C Intermediate Language) [53,54] which translates C language into intermediate language provides instrumentation functions. Pin [55,56] and Valgrind [57,58] are capable of instrumenting binary files in runtime.

A symbolic value is any expression e in some theory T where all free variables are exclusively input parameters. For any program variable v, $M(v)$ denotes the concrete value of v in M, while $S(v)$ stands for the symbolic value of v in S. $S(v)$ is always assumed to be defined and $S(v)$ is simplified as $M(v)$ by default if no symbolic expression in terms of inputs is associated with v is S. Program variable v is symbolic when $S(v)$ differs from $M(v)$, indicating that the value of program variable v is a function of some input(s) which is represented by symbolic expression $S(v)$ associated with v in the symbolic store. We use the notation $M(e)$ to denote the concrete value of symbolic expression e when evaluated with the concrete store M. The notation + for mappings denotes updating. For example, $M' = M + [m \rightarrow e]$ is the same map as M, except $M'(m) = e$.

The program P operates memory (concrete and symbolic) through statements or commands that are abstractions of the machine instructions actually executed. Three kinds of command are defined. An assignment statement **a** is of the form $v := e$ (where v is a program variable and e is an expression); a conditional statement **c** is of the form if e then C_1 else C_2 (where e denotes a Boolean expression, and C_1, C_2 denote the next command to be evaluated when e holds or not respectively); a **stop** statement represents a program error or normal termination.

Given an input vector I with a concrete value I_i to the ith input parameter, the execution of program defines a unique finite program trace $s_0 \xrightarrow{c_1} s_1 \ldots \xrightarrow{c_n} s_n$ that executes the finite sequence

```
1: void example1(int x, int y, int z)
2: {
3:   int cnt=0;
4:   if (x == 1) cnt++;
5:   if (y == 2) cnt++;
6:   if (z == 3) cnt++;
7:   if (cnt == 3) abort(); // error!
8: }
```

Fig. 1. A simple C program to describe how to generate tests automatically by means of DSE. Eight purposive test cases can systematically explore all feasible execution paths of example1. The error in statement 7 can be triggered when the variable *cnt* equals 3.

$C_1 \ldots C_n$ of commands and goes through the finite sequences $s_1 \ldots s_n$ of program states. Each program state is defined as a tuple $\langle C, M, S, pc\rangle$ where C is the next command to be executed and pc is a special meta-variable that represents the current path constraint. A path constraint pc_w for a finite sequence w of commands is a formula of theory T that characterizes the input assignments for which the program executes along w. For convenience, all the program variables are assumed to have some unique initial concrete value in the initial concrete store M_0 and the initial symbolic store S_0 consists of the program variables v whose values are program inputs. Initially, pc is defined as true.

The objective of DSE is systematically exploring all feasible paths of the program under test by using path constraints and a constraint solver. A constraint solver is an automated theorem prover that is capable of judging whether the path constraint is satisfiable. Given a program state $\langle C, M, S, pc\rangle$ and a constraint solver for theory T, if C is a conditional statement in the form defined above, any satisfying assignment to the formula $pc \wedge e$ will lead the program to run the *then* branch of the conditional statement and any satisfying assignment to the formula $pc \wedge \neg e$ (where $\neg e$ denotes the negation of expression e) will guarantee the program to exercise the *else* branch. All possible path constraints can be enumerated and eventually all feasible program paths can be exercised by systematically repeating this process.

The search is exhaustive provided that the generation of path constraints and the solver are both *sound and complete* (see [11] for further proofs), i.e., for each program path w, the constraint solver return a satisfying assignment for the path constraint $\mathbf{\Phi}_w$ *if and only if* the path w is *feasible*. Therefore, in addition to finding errors, an exhaustive search can also prove their absence, thus obtain a form of *verification*.

We give a simple example (Fig. 1) above to illustrate how to generate tests automatically by means of DSE. In the first run, input parameters are generated randomly and the inputs can be captured by instrumentation tools in runtime. Then concrete inputs will be symbolized, i.e., x, y, z will mapping to symbols "input(0)", "input(1)", "input(2)" respectively. Without losing generality, comparisons in statements 4, 5, 6 are assumed to false, so we obtain the path constraint for the first run $\mathbf{\Phi}_1 = \langle \text{input}(0) \neq 1, \text{input}(1) \neq 2, \text{input}(2) \neq 3\rangle$.

To force the program execute a different path, one can calculate a solution for a different path constraint, say, $\langle \text{input}(0) \neq 1, \text{input}(1) \neq 2, \text{input}(2) = 3\rangle$ which is obtained by negating the last predicate of the current path constraint $\mathbf{\Phi}_1$. A new test case expressed by the solution can make the comparison in statement 6 true. By repeating this process, all the eight feasible program paths can be exercised. And the test case $x = 1, y = 2, z = 3$ solved from the path constraint $\langle \text{input}(0) = 1, \text{input}(1) = 2, \text{input}(2) = 3\rangle$ is able to trigger the error in statement 7.

3. Remarkable advantages of DSE

Before people are aware of the advantages of DSE, static techniques and black-box fuzzing are the mainstream that have been used in many security tools, e.g., [59–61]. Now, DSE is becoming a hot research point. The reason lies in that DSE is far more accurate than static analysis and it has a better code coverage compared to black-box fuzzing.

```
1: void example2(int x, int y)
2: {
3:   if (x == hash(y)) abort(); // error()!
4: }
```

Fig. 2. An example to illustrate the limitation of static test generation. An error can be triggered in statement 3 when the input value x equals the calculation of $hash(y)$. Yet, static techniques cannot generate two input values x and y that are guaranteed to trigger this error.

Static test generation is often ineffective.

The major difference between static test generation and DSE lies in that static techniques analyze a program P without actually executing it. Yet static analysis can also employ the symbolic execution technique in order to steer P executes along specific paths or branches. The core idea of static test generation is to symbolically explore all feasible *control paths* of the execution tree when all possible input values are considered. For each control path ρ, i.e., a sequence of control locations of program P, a path constraint $\mathbf{\Phi}_\rho$ is constructed to characterize the input values for which the program executes along ρ. All program paths can be enumerated by a search algorithm that explores all possible branches at conditional statements.

However, static techniques often ineffective, especially whenever the control path contains extern/complex arithmetic functions. In this case, constraint solvers are not capable of reasoning about those functions symbolically. This problem is depicted by the following example (Fig. 2). Given that the function *hash* is an external function or a complex arithmetic function, that is, *hash* cannot be reasoned about symbolically. Therefore, it is in general impossible to generate input values that are guaranteed to trigger the error in statement 3. Unfortunately, issues like this example are very common due to complex program statements (pointer operations, arithmetic computations, etc.) and calls to operating system and library functions, so static test generation is doomed to perform poorly.

DSE is more powerful than static test generation. The problems resulting from extern/complex arithmetic functions can be alleviated with the help of runtime information of program P. Consider again the example above. In the first run, we generally cannot generate the input values x and y to trigger the error, but we collect the calculation of $hash(y)$ dynamically. In the next run, we can set new input x that is equal to $hash(y)$ by fixing the former y so as to force the execution toward the *then* branch.

Black-box fuzzing always has low code coverage.

Black-box fuzzing, which automatically generates random input values and which is able to find bugs in many applications, is attractive mainly because it requires little manual work [62,63]. However, black-box fuzzing has a severe drawback that it misses errors triggered by narrow ranges of inputs since it is blind to the internal execution of program P. Therefore, the code-coverage of black-box fuzzing is doomed to low. This limitation can also be illustrated by the example in Fig. 1. Given that the input values x, y and z are all 32-bit integers, the probability that in one run, we generate x, y and z that are guaranteed to equal to 1, 2, 3 respectively is astronomical small ($1/2^{96}$). That is, black-box fuzzing will most likely miss the error in statement 7 in reasonable time.

On the contrary, at most eight purposive test cases (eight is the upper limit, actual number of test cases needed to reveal the error depends on the search algorithm) generated by DSE can guarantee to hit the error. In DSE, concrete execution is side-by-side with symbolic execution and path constraint is gathered in runtime, thus DSE can obtain necessary runtime information to guide the

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

execution of program under test toward uncovered code. Hence, DSE is sometimes considered as white-box fuzzing [2].

In recent years, a great deal of research attempts to give guidance to black-box fuzzing so as to gain higher code-coverage, though it entails manual intervention. A typical guidance is manually-written language grammar [64,65] that helps black-box fuzzing tools generate legal and illegal input values. However, due to the burdensome manual work for building guidance, very few real applications under black-box fuzzing can be fed with well-structured inputs.

4. Challenges and solutions

To the best of our knowledge, mature tools based on dynamic symbolic execution for commercial use do not exist yet, though this technique has been investigated for a few years. The majority of current DSE tools such as Fuzzgrind [1], KLEE [8], EXE [10], DART [11], CUTE [12] are only for research purpose. SAGE [2], a commercial tool, which is developed by Microsoft, is not available outside of Microsoft Corporation. Pex [38], a DSE tool for unit testing which works as a plug-in component of Visual Studio, aims at.NET program only. Those typical DSE tools are compared and analyzed in Section 6. The reason for the under-deployed of DSE is that several severe challenges hinder the excellent idea against turning into reality. In this section, we describe the challenges and the state-of-the-art solutions in detail.

4.1. Challenge 1: path explosion

Path explosion denotes that the number of feasible execution paths of a program increases exponentially with the increase in the length of an execution path [37], thus exploring all feasible paths is typically prohibitively expensive. Path explosion is the main obstacle against the widespread application of DSE [66]. In recent years, several solutions have been proposed in order to alleviate path explosion. Although existing techniques accelerate DSE for several orders of magnitude, path explosion is still a severe problem since modern software has enormous or even infinite feasible paths. In this section, we describe the current solutions against path explosion in detail.

Intelligent search strategies.

Depth-first search, which employs standard depth-first search algorithm to explore the execution tree, usually appears in early DSE tools, such as EXE [10] and DART [11]. Albeit simple, depth-first search has a fatal shortcoming: it can get stuck in non-terminating loops with symbolic conditions if no maximum depth is specified. When the depth-first search gets stuck, no new code can be executed and the code-coverage is bound to low. Several security tools, e.g., [8,10], adopt a straightforward method that setting a maximum depth in order to prevent infinite loops. Given a test case, whenever a run of symbolic execution violates the maximum depth, this run will be forced to terminate. Yet this method seems too aggressive that it may miss some code which could be exercised by original DSE especially the maximum depth is set too small.

Generational search [2], proposed in 2008, which is the preferred search strategy of the subsequent research, has been widely used in nowadays DSE tools e.g., [1,2,38]. Generational search has four merits that alleviate path explosion. First, it is capable of exploring execution tree of large applications with large input space and with very deep paths. Second, it maximizes the number of new tests generated from each run of DSE, while avoiding any redundant test cases in search. Third, it maximizes code-coverage as quickly as possible in order to reveal vulnerabilities faster. Fourthly, it is resilient to divergences, i.e., whenever divergences occur, generational search is able to recover

```
1: void Search(inputSeed)
2: {
3:   inputSeed.bound = 0;
4:   workList = {inputSeed};
5:   Run&Check(inputSeed);
6:   while(workList is not empty)
7:   {
8:     input = PickFirstItem(workList);
9:     childInputs = ExpandExectuion(input);
10:    while(childInputs is not empty)
11:    {
12:      newInput = PickOneItem(childInputs);
13:      Run&Check(newInput);
14:      Score(newInput);
15:      workList = workList + newInput;
16:    }
17:  }
18:}
```

Fig. 3. Generational search algorithm [2].

```
1: ExpandExecution(input)
2: {
3:   childInputs = {};
4:   PC = ComputePathConstraints(input);
5:   for(j = input.bound; j < |PC|; j++)
6:   {
7:     if((PC[0..(j-1)] and not(PC[j]))
                 has a solution I)
8:     {
9:       newInput = input + I;
10:      newInput.bound = j;
11:      childInputs = childInputs + newInput;
12:    }
13:  }
14:  return childInputs;
15:}
```

Fig. 4. Computing new children of generational search algorithm [2].

and continue. In the context of symbolic execution, divergence denotes that an actual execution path does not match the path predicted by a constraint solver. Due to the importance of generational search, it is necessary to illustrate it in more detail through the following algorithm [2].

The main *Search* procedure (Fig. 3) is mostly standard and the main originality of generational search is in the way of computing new children procedure which is shown in Fig. 4. This algorithm attempts to expand every constraint in the path constraint (at a position j larger or equal to a parameter called *input.bound*), instead of just the last one in the depth-first search. Hence, it maximizes the number of new test cases generated from each symbolic execution. The parameter *bound* is used to limit the backtracking of each sub-search above the branch where the sub-search started off its parent, so redundant search can be avoided.

The function *Score* that is akin to the "best-first" search strategy of EXE [10] and to the "coverage-optimized" search strategy used in [8] is used to compute the incremental code-coverage obtained by executing the *newInput*. The function *PickFirstItem* always picks the test case with the highest score in *workList* in order to maximize code-coverage as quickly as possible. Generational search is resistant against divergences because when divergences occur, we can select another test case for subsequent symbolic execution. In contrast, depth-first search is vulnerable to divergences, since at most one new test case is generated from one symbolic execution.

KLEE [8] uses the two search strategies: coverage-optimized search and random path selection in a round robin fashion in order to achieve high code-coverage. Random path selection can be illustrated as follows: execution paths are selected by traversing the execution tree from the root and randomly selecting the path to follow at branches. Therefore, when a branch is hit, all paths that have not reached their leaves have equal probability to be selected, regardless of the size of their subtrees. Random path selection seems simple, whereas it has two significant advantages. First, it favors paths with fewer constraints and so it has greater freedom to hit uncovered code. Second but more important, it avoids starvation since execution paths with large numbers of

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

```
1: void example3(int x)
2: {
3:   if (x > 0) return 1;
4:   return 0;
5: }
```

Fig. 5. An example to describe function summary. The summary of function *example*3, $(x > 0 \wedge ret = 1) \vee (x \leq 0 \wedge ret = 0)$, can be computed in two runs.

```
1: void example4(int s[N]) // N inputs
2: {
3:   int i,cnt = 0;
4:   for(i = 0; i < N; i++)
5:     cnt = cnt + example3(s[i]);
6:   if(cnt == 3) abort(); // error!
7: }
```

Fig. 6. An example to explain how function summary allay path explosion. Without function summary, 2^N runs are needed to explore all paths of example4, while leveraging summary, merely four runs (two for computing function summary of example3 and two for executing both branches in statement 6) are guaranteed to trigger the error in statement 6.

branches are not able to predominate execution chances with the random selection. By interleaving the two strategies, KLEE can continue where an individual strategy gets stuck. Moreover, interleaving them can improve overall effectiveness [8].

Pex [38] devises a meta-strategy to avoid getting stuck in a particular area of the program by a fixed search order. Pex makes a fair choice between all unexplored branches that it partitions all branches into equivalence classes and then picks a representative of the least often chosen class. All branches in Pex are categorized by the branch coverage, depth and so on.

Summary.

Summary is an indispensable resort to alleviate path explosion [41]. Summary is defined as follows. For a given theory T of constraints, a *function summary* Φ_f for a function f is defined as a formula of propositional logic whose propositions are constraints expressed in T, i.e., Φ_f is defined as a disjunction of formulas Φ_w, $\Phi_w = pre_w \wedge post_w$, where pre_w is a conjunction of constraints on the inputs of f while $post_w$ is a conjunction of constrains on the outputs of f. The definition may confuse readers, so we give a simple example (Fig. 5) to clear the fog.

Given that, in the first run, input value x is greater than zero, so the *then* branch in statement 3 can be hit. Along with the execution path $w1$ of the first run, pre_{w1} and $post_{w1}$ are gathered that pre_{w1} is expressed by $x > 0$ and $post_{w1}$ is expressed by $ret = 1$. Hence, Φ_{w1} is $(x > 0 \wedge ret = 1)$. In the next run, which is guaranteed to execute statement 4, we obtain a formula Φ_{w2} that $\Phi_{w2} = (x \leq 0 \wedge ret = 0)$. As a result, the summary of function *example*3, which is a disjunction of Φ_{w1} and Φ_{w2}, $(x > 0 \wedge ret = 1) \vee (x \leq 0 \wedge ret = 0)$, can be computed in merely two runs.

Here, we use an example (Fig. 6) to explain how to alleviate path explosion through function summary. In this example, two runs are needed to explore all paths of example3, so there are totally 2^N paths of example4 should be executed symbolically without the help of function summary. While leveraging function summary, merely four runs are guaranteed to trigger the error in statement 6. In which, two runs are used to compute summary of example3 and the other two runs execute both branches of statement 6 by solving the constraint path: $[(s[0] > 0 \wedge ret_0 = 1) \vee (s[0] \leq 0 \wedge ret_0 = 0)] \wedge [(s[1] > 0 \wedge ret_1 = 1) \vee (s[1] \leq 0 \wedge ret_1 = 0)] \wedge \cdots \wedge [(s[N-1] > 0 \wedge ret_{N-1} = 1) \vee (s[N-1] \leq 0 \wedge ret_{N-1} = 0)] \wedge (ret_0 + ret_1 + \cdots + ret_{N-1} = 3)$.

Loops in the tested programs are so notorious that DSE is prone to get stuck in them. Patrice and Daniel proposed a powerful technique so called loop summary [42] which is able to alleviate path explosion arose by loops. In theory, loop summary is similar with function summary that it is a logic formula of the form $pre_{loop} \wedge post_{loop}$ where pre_{loop} is a loop precondition defining a set of executions covered by the summary and $post_{loop}$ is a loop postcondition capturing side-effects occurring during those executions. Just as function summary, the total path needed to explore can be drastically reduced by loop summary.

Therefore, the conclusion is that summary is able to reduce the number of paths needed to explore for several orders of magnitude. However, the application of summary is restricted to some specific cases in practice for three reasons. First, summary is usually computed manually [5]. Second, precise function summaries of practical functions are hard to obtain since practical functions always have complex internal program logic (may be incomprehensible by SMT solvers) and have large numbers of feasible paths. Besides, loop summaries computed by the method proposed in [42] may not be complete. In other words, loop summaries perhaps cover partial paths of loops.

Input space reduction.

For convenience, in this paper, we use input space reduction to denote symbolic grammar, length abstraction and information partition. When programs under test have highly-structured inputs, e.g., compilers and interpreters, DSE suffers from the similar problem with black-box fuzzing that it rarely reaches parts of the application beyond the first stages such as lexing, parsing and evaluation that have enormous number of control paths.

Symbolic grammar which is mirrored from black-box fuzzing is an efficient manner to address this issue. Godefroid et al. [43] and Majumdar et al. [44] proposed their *grammar-based* DSE tools which enhance symbolic execution with a grammar-based specification of valid inputs. By restricting input space to valid inputs, grammar-based fashion can exercise deeper paths and focus the search on the harder-to-test, deeper processing stages. Furthermore, symbolic grammar also helps to allay path explosion since a large number of branches in early stages that need not be exercised can be avoided.

Here, we only present the method so called CESE proposed in [44] and the other method can be found in [43]. The first step is building symbolic grammar from concrete grammar. In this step, three heuristic rules are applied in constant string, finite regular language and infinite regular language respectively. Once a symbolic grammar is constructed, several exhaustive enumeration techniques [50,67,68] are applied to generate strings. Then symbolic execution is performed on the instrumented program.

Grammar-based symbolic execution indeed narrows the input space, yet it has a significant drawback that grammar is usually constructed manually. Actually, manual grammar construction that is expensive and burdensome is the main obstruction against widespread application of this technique.

Xu et al. presented a security tool named *splat* [45] that performs DSE for buffer overflow testing. Compared to primitive DSE, splat only tracks a prefix of the buffer symbolically, and a symbolic length that may exceed the size of the symbolic prefix, instead of treating the entire contents of an input buffer as symbolic. This method, termed by *length abstraction*, is able to alleviate path explosion and thus speed up symbolic execution since it only tracks the influence of date values stored in the prefix of input buffer, instead of full buffer. The intuition is that symbolically tracking the entire contents of input buffer is always more precise than necessary for finding vulnerabilities, i.e., many memory errors only depend on the size of input buffer, instead of its content.

Splat consists of three core modules: CIL source-to-source instrumenter [53], a library for tracking memory operations and a library for symbolic execution. In order to abstract buffer length, Splat symbolizes it. Only a short prefix of the buffer is traced symbolically while the characters beyond the prefix are randomly generated. Length abstraction, by definition, can reduce input

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

space dramatically, yet it is an underapproximation approach, so false negatives are inevitable. Moreover, length abstraction is specially designed for discovering buffer overflow vulnerabilities, so it is not a general solution against path explosion.

Majumdar and Xu proposed a novel approach [46] termed by input partition to reduce input space that dividing the entire input space into "non-interfering" blocks and symbolically solving for each input block while keeping all other blocks fixed to concrete values. The efficiency of DSE can be drastically raised by this technique meanwhile it is able to find the same errors with primitive DSE. The main originality is an automatic test generation algorithm *FlowTest* which is an extension of DSE. The key difference between FlowTest and the primitive DSE lies in that FlowTest computes control and data dependencies among variables dynamically and uses these dependencies to keep independent variables separated during test generation.

In the best cases, the number of inputs can be reduced from exponentially many to linearly many. However, FlowTest still has two drawbacks. First, false positives could be produced by defective dependency analysis, especially when analyzing the artificial dependencies caused by error handling code. Second, FlowTest is not general enough to cope with all programs. Because in many programs, some parts of inputs are relevant to all other inputs, however FlowTest performs well only when the assumption holds: the input space of tested programs can be divided into disjoint sub-spaces.

4.2. Challenge 2: extern/complex arithmetic functions

When encountering extern/complex arithmetic functions, DSE performs better than static test generation as shown in Section 3. However, since the extern/complex arithmetic functions are outside the scope of reasoning of the theorem prover, DSE also suffers from this issue in many cases. For instance, when we substitute the statement 3 in Fig. 2 with the following statement "*if* $(hash1(x) == hash2(y))$ *abort*();", here *hash*1 and *hash*2 are two functions that we cannot reason about, DSE cannot guarantee to trigger this error either.

Concretization is a compromised method to circumvent extern/complex arithmetic functions. Concretization consists of using concrete input values to replace symbolic variables, running the extern/complex arithmetic functions as normal without gathering constraints about those functions. Albeit straightforward and reasonable in specific cases, concretization is underapproximation that results in false negatives.

Summary [39–41] is a more precise yet complex manner compared to concretization. When summaries of extern/complex arithmetic functions are at hand, we can treat those functions as black boxes, i.e. we do not trace into those functions neither we reason about them symbolically meanwhile we do not sacrifice accuracy. However, summary is not impeccable since it typically needs to be generated manually [5].

4.3. Challenge 3: floating-point computations

Many applications tend to use floating-point instructions available on modern computers, e.g., codecs, image viewers and media players. Yet existing security tools [1,2,8–12] based on DSE do not handle floating-point computations. The reason mainly lies in that off-the-shelf constraint solvers, e.g., [69–72], are not capable of handling floating-point arithmetic.

All of those tools mentioned above substitute symbolic variables with concrete values when encountering floating-point instructions in order to execute tested program normally. However, concretization leads to false negatives. An easy-thought solution would be to extend DSE to handle floating-point instructions and extend constraint solvers to reason about floating-point constraints. Yet this solution seems naïve for now because the enhancement of symbolic execution and solvers entails months of manual labor or even longer. More disturbingly, this solution will generate more complex constraints and then give rise to even worse path explosion.

```
1: void example5(int x, int y)
2: {
3:   int s[4];
4:   s[0] = x;
5:   s[1] = 0;
6:   s[2] = 1;
7:   s[3] = 2;
8:   if(s[x] == s[y] + 2)
9:     abort(); // error!
10:}
```

Fig. 7. An example to explain why concretization leads to divergences. Given the first inputs $x = 0$ and $y = 1$, symbolic pointer constraint in statement 8 is concretized as $x \neq 2$. Therefore, new constraint is $x = 2$ and then new inputs are $x = 2$ and $y = 1$. In the next run, a divergence occurs that the statement 9 which should be hit is actually missed [48].

Godefroid et al., in 2010, proposed an alternative approach [47] to alleviate this issue. This technique is capable of proving memory safety of floating-point computations meanwhile it does not require accurate symbolic reasoning about floating-point code. The core idea is to treat all floating-point values as a single special symbolic value "FP-tag" during DSE and then perform a dynamic taint-based analysis of FP-tags. Whenever a FP-tag is used to compute a memory address which is dereferenced during symbolic execution, an alarm will be reported. The intuitionistic explanation is that floating-point code should only perform memory safe operations of the "payload" of an image or a video, while the normal (non-floating-point) instructions can deal with critical operations which may lead to security problems e.g., buffer allocation and memory address computations.

This novel technique consists of a lightweight local path-insensitive overapproximating static analysis of floating-point instructions and a whole-program path-sensitive underapproximating dynamic symbolic analysis of normal instructions. Preliminary experiments show that this approach is competent to prove memory safety of floating-point computations at the cost of a small upfront static analysis and a marginal runtime expense, hence avoiding costly floating-point constraints gathering and solving. However, because of the inherent overapproximating property, this taint-based technique suffers from innegligible false positives.

4.4. Challenge 4: symbolic pointer operations

A symbolic pointer operation denotes a pointer dereference at an address whose value depends on the evaluation of a symbolic expression, i.e., depends on some (untrusted) inputs. A straightforward solution against this challenge is concretization, i.e., whenever symbolic execution engine does not know how to generate a symbolic constraint for a program statement that contains symbolic pointer operations, the symbolic variables are substituted by concrete values. However, concretization leads to divergences that the actual execution paths do not match with the paths predicated by constraint solvers. As a result, concretization results in obvious false negatives. We borrow the example (Fig. 7) in [48] to explain why concretization leads to divergences.

Given the first input values $x = 0$ and $y = 1$, when the comparison $s[x] == s[y] + 2$ in statement 8 is executed, the path constraint is concretized as $x \neq 2$. New inputs are computed by solving the new path constraint $x = 2$ which is obtained by negating the previous path constraint in order to reach the error in statement 9. However, when executing with the new inputs $x = 2$ and $y = 1$, the program does not execute the expected path. As a result, a divergence happens and the error in statement 9 is missed.

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

Elkarablieh et al. presented a practical yet precise solution through actively injecting new constraints [73] which represent the snapshot of the memory region associated with memory dereferences and the bound of symbolic addresses to handle symbolic pointer operations [48].

For instance, consider again the example in Fig. 7, instead of concretizing the dereferences operations when encountering statement 8, four new constraints that represent the snapshot of the memory region: $s[0] == x, s[1] == 0, s[2] == 1$ and $s[3] == 2$ are injected into the path constraint. Moreover, two new constraints which represent the bound of symbolic addresses: $\&s \leq \&s[x] < \&s + 4$ and $\&s \leq \&s[y] < \&s + 4$ are also added actively. Associated with the original constraint: $s[x] == s[y] + 2$, any solver can easily solve the path constraint and generate new test case: $x = 3$ and $y = 1$ which is guaranteed to hit the error in statement 9. However, this technique that injects new constraints to original path constraint will worsen path explosion. Moreover, keeping track of all valid memory regions currently allocated and maintaining snapshots require plenty of memory resources.

4.5. Challenge 5: environmental interaction

As a large number of real applications are environmentally-intensive, DSE suffers from challenges in handling code that interacts with its surrounding environments, such as the operating system, the network, or the user. In fact, the execution of program under test not only depends on user's input but relies on its surrounding environments. That is to say, if we cannot simulate environment properly, those execution paths related to the interaction with environment may not be reached. As a result, the code-coverage of environmentally-intensive applications analyzed by DSE is always low.

Cadar et al. tried to simulate environment by redirecting library calls that access it to *models* that understand the semantics of the desired actions well enough to generate the required constraints [8]. All legal values can be returned from these models rather than just a single concrete value. Yet writing models by hand for all environmentally-related library calls is a burdensome and error-prone task.

Database applications should be treated in particular since both input data and suitable database records can influence DSE. Emmi et al. proposed a method to handle the interaction between database applications and database by symbolically tracking the *SQL* queries made along the execution path during symbolic execution and then translating constraints in an *WHERE* clause to appropriate constraints that can be understood by SMT solvers [49]. By solving a path constraint consisting of constraints on input data as well as constraints on *SQL* queries, the code-coverage significantly improves.

However, the scheme of Emmi et al. has two shortcomings. First, it supports Java programs and corresponding database only, thus scaling its implementation to a real enterprise system is a significant engineering effort. Second, the constraint solver proposed in this paper is a trade-off between fast constraint solver and the ability to capture many constraints of practical interest, hence, improving constraint solver is possible in future.

An alternative approach was proposed by Taneja et al. that suitable database records can be obtained through mock databases [50]. A mock database mimics the behavior of an actual database by performing identical database operations on itself. It can be seen that mock database generation is crucial in this technique while the task is rather labor-intensive.

Web applications, say, JavaScript applications, also interact with surrounding environments intensively that they often take many kinds of inputs, e.g., user input from form fields, messages from Web servers, data from concurrently running code in other Web browser windows and keyboard/mouse clicks. Saxena et al. proposed a symbolic execution framework for JavaScript [6]. The main originality of their work is that they divide the entire input space into value space consisting of user date, URL and cross-window communication abstractions, HTTP channels and event space which contains a sequence of events resulting from user actions. This technique combines value space exploration which systematically explores execution paths by dynamic symbolic execution and event space exploration that employs a random exploration strategy to search all event sequences by a self-developed GUI explorer.

Chipounov et al. put forward several relaxed execution consistency models in an attempt to generalize the means of environmental interaction [74,75]. DSE tools often employ either the SC-UE or SC-SE models. By the definition of SC-UE model, DSE tools execute tested programs symbolically while they do not instrument the environment and so the environment code must be executed concretely. SC-UE model may miss feasible paths as it does not explore environment paths exhaustively. On the contrary, the environment is modeled and then both the tested program and the model are executed symbolically. Although SC-SE model is more complete than SC-UE model, the former may result in more severe path explosion. In addition, environment models are generated by manual work which is error-prone and burdensome.

5. Other techniques to improve DSE

Many researchers agree that for the whole cost of DSE, the cost of constraint solving dominates everything else since constraint solving is a *NP-complete* problem [8] whose algorithmic solutions are currently believed to have exponential worst case complexity [76]. In order to apply symbolic execution technique in real complex programs, several methods [8,10,12] that aim at simplifying constraint expressions or reducing queries to SMT solvers have been proposed in recent years.

Constraint caching

Intuitively, the efficient of symbolic execution can be significantly promoted by caching the satisfiability queries transformed from path constraints as well as the solutions since the solutions for the same queries can be directly returned by searching the cache without invoking SMT solvers [10].

Cadar et al. presented a more sophisticated caching strategy termed *counter-example cache* that maps sets of constraints to counter-examples along with a special sentinel used when a set of constraints has no solution [8]. Counter-example cache gains three additional ways to eliminate queries. First, when a subset of a constraint set has no solution, then neither does the original constraint set. For example, the query $\{x > 10 \wedge x < 5\}$ has no solution, neither does the original query $\{x > 10 \wedge x < 5 \wedge y = 0\}$. Second, when a superset of a constraint set has a solution, that solution also satisfies the original constraint set. For instance, $x = 1$ is a solution for the query $\{x > 0 \wedge x < 5\}$, thus it satisfies either $\{x > 0\}$ or $\{x < 5\}$ individually. Third, when a subset of a constraint set has a solution, it is likely that this is also a solution for the original set. The intuition of the last heuristic is that extra constraints often do not invalidate the solution to the original constraint set. Besides, checking a potential solution is much cheaper than invoking SMT solvers.

Constraint independence optimization

Constraint independence optimization [8,10] consists of dividing the entire constraints set into several independent constraints subsets and solving them separately. Two constraints are considered as independent if they have disjoint sets of operands. For example, a constraints set obtained from a run of symbolic execution $\{x_1 = x_2 + x_3 \wedge x_2 > 10 \wedge x_4 < 5\}$ can be divided into $\{x_1 = x_2 + x_3 \wedge x_2 > 10\}$ and $\{x_4 < 5\}$. The intuition of this technique is that solving independent constraints subsets is much faster than

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

solving the whole constraints set. Furthermore, this technique promotes the benefits from constraint caching since short constraints subsets increase the probability of hitting the cache.

DomainReduce optimization.

Erete and Orso [77] found that restricting the domain for path condition may help the constraint solver to find a solution faster than when considering the complete input domain. At first, all target symbolic variables but one are fixed to their concrete values. If the solver can find a solution, the solution is also for original path condition. If not, a different symbolic variable will be tried and eventually the domain will be expanded by adding one variable to the set of symbolic variables.

DomainReduce leverages a tradeoff related to the size of the input domain considered. Generally speaking, the smaller the domain, the faster the solver can find a solution, if it exists, but the lower the probability for a solution to exist. Actually, expanding input domain increases the probability to find a solution but significantly raises the cost to find such solution. Albeit this technique seems lacking of theory validation, it performs pretty well in practice.

Expression rewriting.

Expression rewriting [8] is a basic optimization mirrored from that in a compiler. The goal of expression rewriting is to transform the original constraint expression into a much easier form with respect to SMT solvers so as to speed up constraint solving. For instance, $x \times 2^n$ and $2 \times x - x$ are usually simplified as $x \ll n$ and x respectively before sending to constraint solvers.

Constraint set simplification.

Symbolic execution typically involves the addition of a large number of constraints to the path condition. The structure of programs means that constraints on same variables tend to become more specific [8]. For example, given a running program under test with a path constraint $\{x < 10\}$, in a while, a new constraint $\{x = 5\}$ is added. Then previous constraint $\{x < 10\}$ is simplified as *true*. A similar approach named *common sub-constraints elimination* was introduced in [12]. Consider again the example above, the constraint $\{x < 10\}$ is eliminated because it is the sub-constraint of $\{x < 5\}$.

Fast unsatisfiability check, also proposed by CUTE [12], can be considered as an immediate extension of constraint set simplification. Whenever new path constraint is generated by negating specified constraints, fast unsatisfiability check is invoked. It works as follows: if any constraint in the new path constraint syntactically conflicts against other constraints, the result *unsatisfiable* is immediately returned without invoking SMT solvers. For example, a new path constraint $\{x_1 > 1, y = 10, x_1 < 0\}$ which is generated by negating the last constraint of previous path constraint $\{x_1 > 1, y = 10, x_1 \geq 0\}$, is unsatisfiable since the constraint $\{x_1 < 0\}$ conflicts against the first constraint $\{x_1 > 1\}$. Therefore, without invoking SMT solvers, we know the new path constraint is unsatisfiable. Fast unsatisfiability check is much cheaper than invoking constraint solvers, hence overall effectiveness of DSE is improved.

Implied value concretization.

When a new added constraint implies a concrete, the entire path constraint can be simplified by substituting the symbolic variable with the specified concrete value [8]. For instance, given a running program under test with a path constraint $\{x + y > 10, x + z < 0\}$, in a while, a new constraint $\{x = 1\}$ is gathered. The implication of the first and the second constraint are $\{y > 9\}$ and $\{z < -1\}$ respectively. Therefore, in this example, the entire path constraint can be simplified as $\{y > 9, z < -1, x = 1\}$.

Incremental solving.

The intuition of incremental solving is that the path constraints $\mathbf{\Phi}_w$ and $\mathbf{\Phi}_u$ from two consecutive symbolic executions differ in a small number predicates (more precisely, only in the last predicate when employing depth-first search), and thus the respective solutions I_w and I_u must agree on many mappings. CUTE [12] exploits this property by incrementally solving path constraints.

At first, CUTE constructs a constraint set D consisting of predicates in previous path constraint $\mathbf{\Phi}_w$ that are dependent on the predicates negated for solving new test cases. The solver then finds a solution I_D for the conjunction of all predicates from D. The new test case I_u for the next run is $I_w[I_D]$ which is the same as I_w except that for every k for which $I_D(k)$ is defined, $I_w(k) = I_D(k)$. Incremental solving is capable of drastically speeding up constraints solving since the size of D is almost one-eighth the size of normal path constraint in practice. A similar approach could be found in [78,79].

Loop counts symbolization.

Some code could be triggered only when the iteration numbers of specific loops reach any specified values. In practice, this type of code is commonplace. For example, a bug could be triggered only when the length of input exceeds a specified value, to be more accurate, the iteration number of the loop which copies the input into a buffer is larger than the length of buffer. Otherwise, the bug cannot be discovered no matter what content of the input is. However, traditional DSE can rarely execute such type of code since the iteration numbers are treated as concrete values.

Saxena et al. figured out this issue by introducing new symbolic variables to represent the number of times each loop in the program has executed [80]. Two steps are required to complete the task. First, the linear relationship of loop counts and loop dependent variables is discovered. Then the relationship of loop counts and the input is built. As a consequence, the loop dependent variables could be set to proper values which can trigger some specified code by changing the input.

6. Comparison and analysis of typical DSE tools

In this section, we compare and analyze twelve typical DSE tools from many aspects. Researchers and practitioners can indeed benefit from this section for choosing proper DSE tools for their own purposes, if they exist. Otherwise, this section helps people build novel DSE tools by referring exist tools. Detailed comparison results are presented in Table 2.

One thing we have to point out is that the tools listed here are not complete, actually only twelve well-known tools are presented. Additionally, the items appeared in Table 2 are referred to public-available materials such as academic papers, technique reports, websites etc rather than any proprietary materials and source code etc. Hence, the contents of Table 2 are not guaranteed to be accurate. For example, two optimization techniques of one DSE tool are presented in Table 2 according to its articles but in fact the DSE tool possesses more optimizations. One point worth mentioning is that we do not attempt to compare those tools with regards to their abilities, soundness, effectiveness etc because they run on different platforms, use different instrumentation tools and different constraint solvers as well as have different tested subjects. As a consequence, it is not quite possible to make direct comparison on their performance at this moment.

The first column lists the names of DSE tools in comparison. The platforms that the DSE tools can run on are presented in column two. Column three gives the instrumentation tools which the DSE tools rely on. The constraint solvers used in those tools are in column four. No such tools have ability to cope with all types of tested programs and so column five shows the types that each DSE tool can analyze. Column six presents the path selection algorithms adopted by different tools. The methods to handle the interaction between tested programs and their environments are shown in column seven. Column eight presents the optimization techniques

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

Table 2
Comparison details of twelve typical tools.

Year	2008	2008	2006	2008	2005	2005	2009	2007	2011	2008	2010	2008
Institution	Microsoft Corp.	Microsoft Corp.	Stanford Univ.	Stanford Univ.	Bell Lab.	UIUC	Sogeti/ESEC	UC Berkeley	EPFL	UCLA	Peking Univ.	UC Berkeley
Open source	No	No	No	Yes	No	No	Yes	Yes	Yes	No	No	Partial
Public access	No	Yes	No	Yes	No	No	Yes	Yes	Yes	No	No	Yes
Optimization	1, 2, 4, 7	1, 2	1, 2	1, 2, 3, 4, 5	/	4, 6	4	/	1, 2, 3, 4, 5	8, 9	/	4
Environment modeling	SC-UE	SC-SE	SC-UE	SC-SE	SC-UE	SC-UE	SC-UE	SC-UE	Mixture of six models	SC-UE	SC-UE	SC-UE
Path selection	Generational	Meta-strategy	Depth-first, best-first	Random, coverage-optimized	Depth-first	Depth-first	Generational	Depth-first	Random, depth-first, breadth-first, max-coverage	Depth-first	Depth-first	Breadth-first
Tested program	Binary	.Net program	C program	C program	C program	C program	Binary	Binary	Binary	C program	Binary	Binary
Constraint solver	Disolver [82]	Z3 [69]	STP [70]	STP	lp_solver [85]	lp_solver	STP	STP	STP	STP	STP	STP
Instrumentation method	iDNA [81]	.Net profiling API [83]	CIL [53]	LLVM [84]	CIL	CIL	Valgrind [57]	Valgrind	QEMU [87]	CIL	Pin	TEMU
Platform	Windows	Windows	Linux	Linux	Linux	Linux	Linux	Linux	Windows, Linux, Mac OS	Linux	Windows, Linux	Windows, Linux
DSE tool	SAGE [2]	PEX [38]	EXE [10]	KLEE [8]	DART [11]	CUTE [12]	Fuzzgrind [1]	CatchConv [86]	S^2E [75]	Splat [45]	TaintScope [7]	BitBlaze [24]

1: Caching
2: Constraint independence optimization
3: Expression rewriting
4: Constraint set simplification
5: Implied value concretization
6: Incremental solving
7: Function summary
8: Length abstraction
9: Information partition.

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

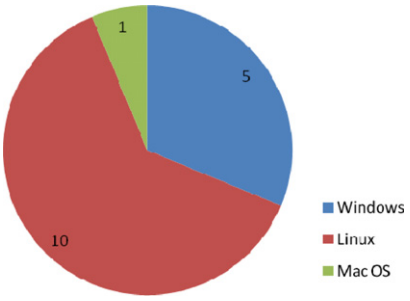


Fig. 8. Statistics of platforms which DSE tools run on.

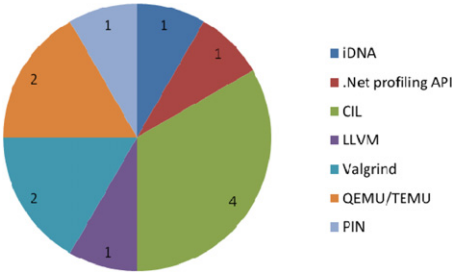


Fig. 9. Statistics of instrumentation tools.

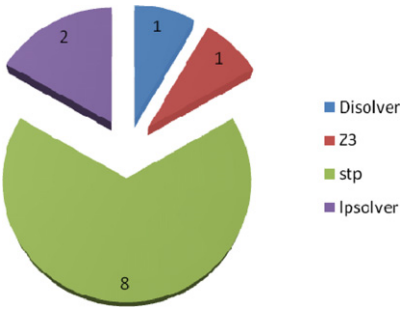


Fig. 10. Statistics of constraint solvers.

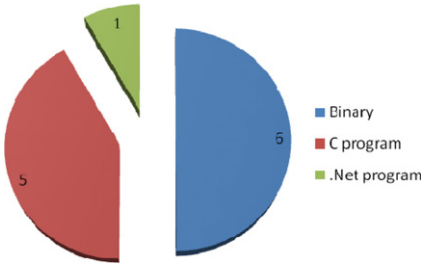


Fig. 11. Statistics of tested programs.

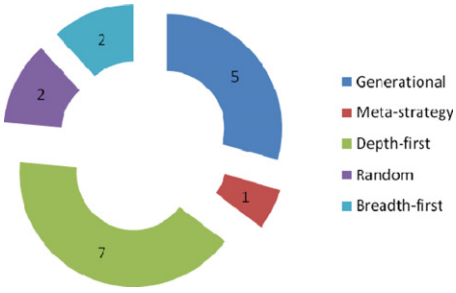


Fig. 12. Statistics of path selection strategies.

applied by different tools. Column nine and ten indicate whether those tools are publicly available or open source respectively. Column eleven reveals which institutions invent those tools. Although several tools are built by several collaborators, we just list the first inventor here. The last column shows when those tools are proposed.

Platforms statistics.

Fig. 8 reveals that nearly two-thirds DSE tools are able to execute on Linux, five of them can run on Windows, and only one can port to Mac OS. One observation from Table 2 is that all but three tools are designed for unique platform. And two out five which support Windows are built by Microsoft Corp. itself. Besides, it can be seen that Linux is the preferred platform by non-profit organizations as all but one institutions which support Linux are universities or laboratories.

Instrumentation tools statistics.

Fig. 9 shows that the instrumentation tools used by different DSE tools vary a lot. As TEMU is built on top of QEMU, we categorize them into one group in Fig. 9. The most-applied one is CIL [53] which occupies exactly one-thirds of total. Besides, Valgrind is the second popular tool which is used by Fuzzgrind and CatchConv.

Constraint solvers statistics.

A conclusion could be made from Fig. 10 is that STP is the most-used constraint solver which makes up of exactly two-thirds of the total. Open source may be one reasonable explanation to its monopoly. But in fact, few DSE tools modified the source of STP to make it better. And STP is not the best constraint solver in terms of performance, function etc. So the inherent reasons for the prevalence of STP are still needed to explore.

Tested programs statistics.

Fig. 11 shows that exactly one half DSE tools are capable of analyzing binaries, five others can handle C programs, and only one supports.Net programs. In fact, the form of tested programs depends on the instrumentation tools selected by different DSE tools. To be specific, Valgrind, Pin, QEMU, TEMU and iDNA support binary executables, CIL and LLVM can handle C programs only and.Net profiling API is designed for.Net programs.

Path selection statistics.

From Fig. 12 we find that depth-first is the most-adopted path selection strategies. An obvious merit of depth-first method is easy-to-implement. But from effectiveness aspects, it is not a good choice in many occasions. The second widely-used method is generational algorithm. As generational algorithm, max-coverage strategy, best-first method and coverage-optimized approach share a lot of similarity, we group them in one category. Experiments indicate that there is no one method can perform well in all situations, so EXE, KLEE and S^2E integrate different strategies in order to gain better flexibility.

Environment modeling statistics.

Fig. 13 reveals that exactly three-fourths DSE tools apply SC-UE model and only two out of twelve uses SC-SE model. The results are reasonable for the following explanation. Although SC-SE model is more accurate than SC-UE model, the models are produced by heavy manual work at the risk of more severe path explosion. On the contrary, SC-UE model is more efficient and easy to implement at the cost of moderate loss of accuracy. One point worth mentioning is that S^2E, as a comprehensive platform, provides six different models which can be specified by users.

Optimization statistics.

Caching, constraint independence optimization and constraint set simplification, as shown in Fig. 14 are the top three preferred optimizations. Moreover, an observation is that four optimization approaches are only used by their inventors. One thing seems somewhat strange is that although Godefroid et al. claim function summary is an indispensable approach to battle against path explosion [41], this technique appears in SAGE only. A possible explanation is that the generation of function summary is

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

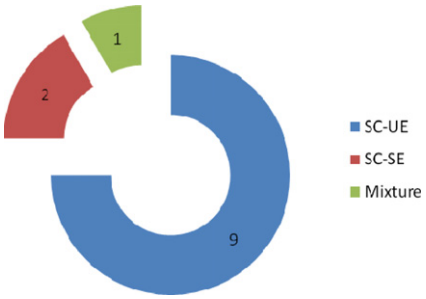


Fig. 13. Statistics of environment modeling.

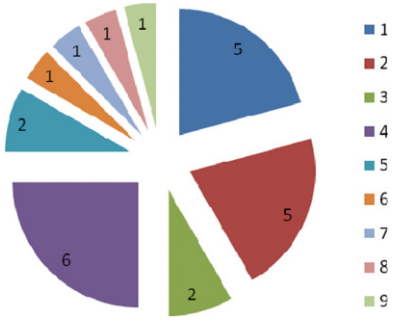


Fig. 14. Statistics of optimization methods.

burdensome and error-prone. From Fig. 14, the conclusion is not so optimistic that the existing optimizations are not widely-deployed in current DSE tools. There is still a long way to improve the performance of those tools.

Public access and open source statistics.

Encouraging information is obtained from Fig. 15 that half of twelve tools are publicly-available and five of them are open source. The two properties make a tight connection between researchers and practitioners from different parts of the world since they are able to communicate and learn from each other in an effortless fashion. Another benefit is that the learning curve of this research field could be significantly moderated by experiencing those downloadable tools as well as analyzing their source code.

Institution statistics.

Fig. 16(a) reveals that exactly three-fourths DSE tools are implemented by non-profit organizations such as universities and laboratories. On the contrary, only three out of twelve are built by corporations. The implication is that this research field is not mature as the industrialization level is fairly low. Fig. 16(b) gives valuable information that USA is the leader of this research field and the other parts of the world act as followers.

Year information statistics

Fig. 17 illustrates the development trend of DSE tools. The first finding is the first two tools were completed in 2005. Then there is a booming development that five tools were accomplished in 2008. Besides, three tools were published after 2008. However, we cannot make a simple conclusion that this research branch is not as hot as that was in 2008. Actually, the comparison presented in the section is not complete that several tools which are not as well-known as the twelve in Table 2 are not listed. In fact, a majority of other tools are built on top of the twelve typical tools mentioned above. For instance, tools proposed in [39,41,51] are extensions of DART, those in [42,43,47,48,88] are based on SAGE, tools introduced in [89–91] are enlightened by KLEE, those brought forward by [40,92,93] are the descendants of PEX, two tools invented in [44,94] are on basis of CUTE, in addition to these, RWset [95] proposed by Boonstoppel et al. is implemented on EXE.

7. Prospects of DSE

In recent years, dynamic symbolic execution has been intensively researched mainly for its two inherent merits: low false positives and high code-coverage. However, hitherto this technique is not competent to handle the real complex programs for the challenges mentioned above. Unfortunately, until now, several challenges e.g., path explosion, floating-point computations and environmental interaction have not been settled properly. Needless to say, these unsolved problems are certain to be the research hot spots in the future. Furthermore, the integration of dynamic symbolic execution and associated research fields, e.g., black-box fuzzing, static analysis, database techniques and parallel computing is also the development trend. In this section, we predict the prospects of DSE for automated test generation in the future.

New techniques to figure out path explosion.

Although there exists many approaches, such as intelligent search strategies, function summary and input space reduction which are able to partly alleviate path explosion, path explosion is still the major obstacle against the widespread application of DSE [66]. The intrinsic reason of path explosion lies in the programs under test since modern software has enormous or even infinite feasible paths. Each existing approach alone is not competent to systematically explore all feasible paths of the real applications. From our standpoint, there are three ways can address path explosion to a great degree.

First, integration of current techniques of DSE is a straightforward improvement. We find that several approaches that aim at alleviating path explosion can complement each other. For instance, we can design a heuristic round robin scheme that alternates in depth-first search, generational search and random path selection to improve code coverage. Moreover, a comprehensive input space reduction strategy can be obtained by combining of symbolic grammar, length abstraction and input partition since the three techniques are orthometric. Furthermore, security tools armed with function summary, constraint caching, constraint independence optimization etc. perform much better than the primitive DSE tools.

Second, hybrid of symbolic execution and black-box fuzzing is also helpful. The research show that black-box fuzzing can reach *deep* branches of the program execution tree by executing a large number of very long program paths quickly. However, black-box fuzzing fails to be *wide*, i.e., to achieve high code-coverage. In contrast, symbolic execution is wide by steering new executions toward new program paths but it is difficult to reach deep branches since maintaining and solving symbolic constraints along execution paths becomes expensive as the length of the executions grow [94]. From the observation, our conclusion is that a hybrid software testing technique outperforms primitive symbolic execution and primitive black-box fuzzing by exploiting their advantages.

Third, collaboration of dynamic techniques with static techniques is worthy to be investigated in depth. Actually, the idea is not novel that a number of techniques which have already applied in DSE are mirrored from static techniques, e.g., function summary [39–41], grammar [43,44] and length abstraction [45]. The intuition is that static analysis that is quick but error-prone and DSE that is accurate but expensive is complementary.

More powerful SMT solvers.

Here, the meaning of powerful is two-fold: fast and versatile. For the whole cost of DSE, the cost of constraint solving dominates everything else since constraint solving is a *NP-complete* problem [8]. Although there are several methods to simplify constraint expressions or reduce queries to SMT solvers, in our opinion, accelerating the constraints solving of SMT solvers selves is a thorough solution. The prospects of accelerating SMT

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

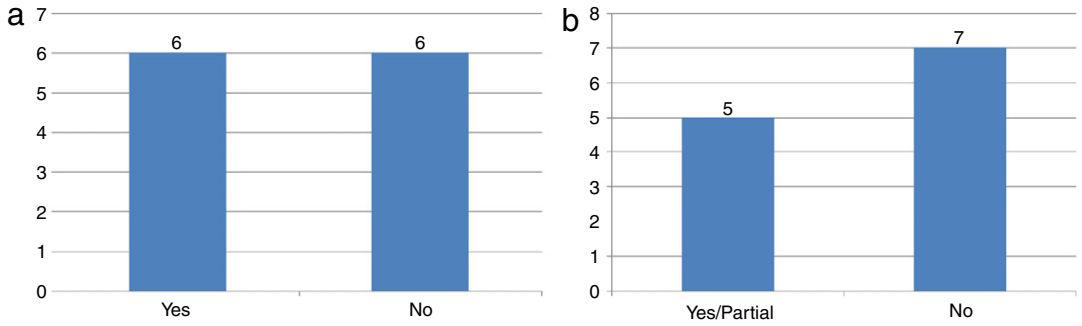


Fig. 15. Statistics of (a) publicly-available and (b) open source.

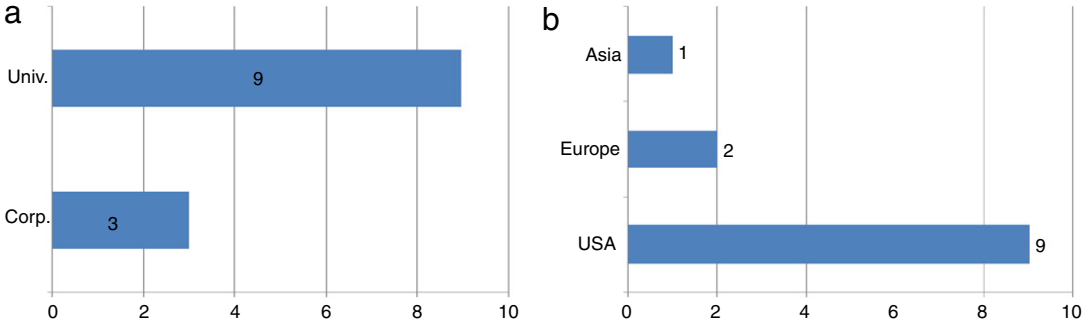


Fig. 16. Statistics of the institution (a) identity (b) district.

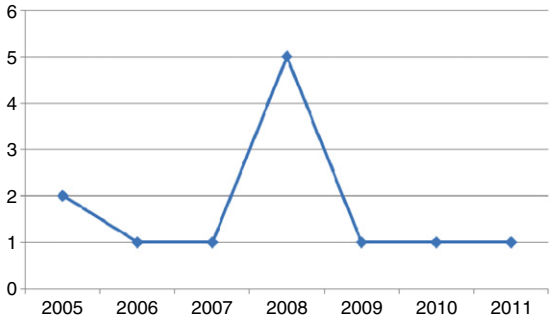


Fig. 17. Development trend of DSE tools.

solvers is promising that SMT solvers have improved significantly reported by the annual Satisfiability Modulo Theories Competition (SMT-COMP) [96] in these years.

Versatility is also an important property of the future SMT solvers. From our viewpoint, future SMT solvers are capable of reasoning about various kinds of constraints, such as arithmetic, bit-vectors, arrays, uninterpreted functions, string operations, floating-point computations, symbolic pointer operations, etc. generated by different applications. To be specific, a number of string constraints are generated by database applications [49] and JavaScript applications [6], so constraint solvers should be optimized to accelerate the solving of string constraints. Besides, future solvers should handle floating-point computations properly since current solutions are defective, e.g., concretization results in false negatives and taint-based method [47] suffers from false positives.

Particular security tools for particular applications.

We believe that DSE will be applied in all areas related to software design/implementation/testing in the future, since software security is receiving more and more importance day by day. From our viewpoint, future security tools based on DSE should have the following properties. First, it must adapt to diverse instruction sets, e.g., X86, PowerPC, SPARC, ARM/StrongARM etc. Note that, different instruction sets vary binary instrumentation tools.

Second, source-based symbolic execution tools should be able to understand and then thus properly instrument source code in different programming languages, e.g., C, C++, VB, Java etc. Hence, we should extend current source-to-source instrumentation tools to cover various programming languages as many as possible. Third, future symbolic execution tools should be able to run on mainstream OS (Operating System) platforms such as Windows, Unix, Linux, VxWorks, PSOS etc. and analysis applications for those mainstream platforms.

We find that incompatible instrumentation tools and SMT solvers are the main obstruction against the transplantation of current security tools to different OS platforms. For instance, Pin [55,56], a binary instrumentation tool, proposed by Intel, can execute on Windows and Linux, while another popular instrumentation tool Valgrind [57,58] is designed for Linux only; Z3 [69], an efficient SMT solver invented by Microsoft, can run on Windows only, while another famous SMT solver STP [70] is particularly for Linux.

Then, future symbolic execution tools should be optimized for different applications, such as desktop applications, backend services, Web applications, database applications etc. The reasons are two-fold. First, we have to symbolically model the surrounding environments of different applications properly. Specifically, we should gather database constraints transformed from *SQL* queries [49]; we have to systematically explore event space when analyzing JavaScript applications [6]. Second, we must optimize SMT solvers for varied kinds of constraints generated by different applications. For instance, database applications [49] and JavaScript applications [6] generate a large number of string constraints (string equality, disequality, as well as membership in regular languages).

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

Furthermore, future DSE tools are able to cope with concurrent systems. Traditional DSE tools arose in the context of checking sequential programs with a fixed number of integer variables, so it is weak in handling concurrent systems with complex inputs [97]. The reason is that DSE does not work in settings where the execution of expressions is not atomic [98]. The research for automatic verification of concurrent systems via DSE is in very early stage. Preliminary studies [99–101] reveal the fact that the collaboration of model checking and DSE is fairly encouraging. Model checking is an automatic and particularly good at analyzing concurrent systems. Meanwhile, the issues of model checking such as state-space explosion, and typically requiring a closed system and a bound on input sizes could be alleviated by DSE [97].

In addition to those above, future DSE tools can handle programs with complex logics and computations. As the ability of DSE tools depends on SMT solvers, they are not capable of analyzing the programs which cannot be expressed by simple logics, such as propositional logic, first-order logic etc. Moreover, current SMT solvers are weak in complex computations, such as non-linear and floating-point arithmetic which makes up-to-data DSE tools fail in such occasions. The research in search-based test generation [29,30,102] convince us that it is a promising technique to help DSE overcome the issue. Theoretically speaking, searched-based test generation relies on search algorithms instead of SMT solvers, so complex logics and computations are not severe problems for it.

However, DSE cannot be substituted by searched-based techniques as the latter has the following drawbacks. First, search algorithms cannot always find solutions, although the solutions indeed exist, because they suffer from local optima and plateaux problems. Conversely, in the problem domain of SMT solver, solutions will always be found if they exist. The second shortcoming is that search algorithms are usually not efficient. Naturally, many attempts will be made before finding a solution, so a lot of redundant test cases will be produced.

Therefore, our conclusion is that designing particular security tools for particular applications is a more realistic undertaking than implementing a comprehensive security tool to cope with all instruction sets, all OS platforms, all programming languages and all kinds of applications because of the arduous tasks mentioned above needed to fulfill.

Manageable summary database.

We agree with Godefroid et al. that summary is indispensable when applying DSE in real complex programs [41]. Summaries are not only re-usable during a fuzzing session, but also apply across applications that share common components (such as DLLs) and over time (from one fuzzing session to the next), so we need a database to store summaries. In order to maximize the benefits obtained from summaries, we should accelerate the interactions between security tools and summary database as quickly as possible.

Besides, with the increasing of the size of summary database, we encounter a severe problem that how to maintain summaries as the code under test evolves slowly [88]. A straightforward but expensive way is recomputing function summary from scratch for every program version. However, it seems wasteful since new versions are frequent and much of the code has typically not changed. So we need a much cheaper way to check the validity of the previous-computed summaries. Anyway, an underapproximating method is not an option since false negatives may disturb symbolic execution. While an overapproximating method with a few false positives is acceptable because false positives can be made up by recomputing summaries.

Integration of DSE and parallel computing.

The complexity and scale of applications could be analyzed by DSE is limited by CPU performance. Although today we can analyze much larger programs than what could do yesterday, our desire is endless that we expect more powerful CPUs to address still larger programs. Sadly, the effect of Moore's curve to drive a continuing increasing in the performance of CPUs appears to be diminishing somewhat sooner than anticipated [103]. In order to obtain substantial computing ability, we have to resort to parallel computing, such as multi-core with shared memory [104], clusters [105], cloud computing [89,90,106] and NVIDIA CUDA GPU [107].

Fortunately, the natural procedure of DSE makes the integration of parallel computing possible. The task of DSE can be considered as exploring the whole feasible paths of the program under test. The research show that the task can be divided into many parallelizable subtasks—each explores a part of whole paths. Note that, the union of paths explored by all subtasks must cover the whole feasible paths. Otherwise, some paths could be missed. Furthermore, the procedure of solving path constraint is also parallelizable. Original satisfiable problem can be divided into many sub-problems. And then those sub-problems are solved in parallel by invoking SMT solvers. At last, the solution of original problem is built by a compilation procedure which joins all solutions of sub-problems.

Thus far, a few preliminary studies have been carried out to introduce parallel computing into DSE. Results are promising that several orders of magnitude improvement is obtained. Load balancing is the major challenge in technique required to be addressed [89,104]. The intuition is that a set of parallel tasks perform most efficiently only if tasks are distributed approximated evenly between task nodes (e.g., CPU cores, computers, GPU cores) and task nodes communicate each other as little as possible.

In order to promote the application of parallel DSE in practice, not only techniques but also markets must be considered. For instance, cloud fuzzing (denotes the integration of DSE and cloud computing) [89,90,106] makes DSE run in the cloud and thus users rent services in the terminals. Hence, a proper pricing scheme is the key to promote cloud fuzzing. Preliminary study [90] shows that cloud fuzzing is economically feasible that the cost of single user is very low since the cost of testing common bodies of code can be amortized over all customers who will need those results later on.

Parallel computing provides substantial computing ability for DSE and their integration is technically and economically feasible, therefore our conclusion is that parallel techniques will fundamentally revolutionize DSE in near future.

8. Conclusion

Dynamic symbolic execution for automated test generation consists of instrumenting and running a program while collecting path constraint on inputs from predicates encountered in branch instructions, and of deriving new inputs from previous path constraint by an SMT solver in order to steer next executions toward new program paths. DSE has been introduced into several applications, such as automated test generation, automated filter generation and malware analysis mainly for two intrinsic properties—low false positives and high code-coverage.

In this paper, we focus on the topics that are closely related to automated test generation. We have five major contributions. First, we summarize the theoretical foundation of DSE including the definition of DSE and the advantages of this technique. Second, we highlight the challenges e.g., path explosion, extern/complex arithmetic functions, floating-point computations, symbolic pointer operations and environmental interaction, when turning ideas into reality. Third, we dwell on the state-of-the-art solutions such as intelligent search strategies, summary, input space reduction etc. to battle against these challenges.

Moreover, we describe the current techniques such as constraint caching, constraint independence optimization, expression rewriting etc. to accelerate symbolic execution based software testing. Furthermore, twelve typical DSE tools are compared and

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

analyzed. Researchers and practitioners will benefit from Section 6 that they can select proper tools for their own sake or make a correct decision to build novel tools from scratch. At last, we predict the prospects of DSE for automated test generation. From our standpoint, solutions for path explosion, powerful SMT solvers, special designed tools for specific situations, Manageable summary database, and the introduction of parallel computing are the future directions.

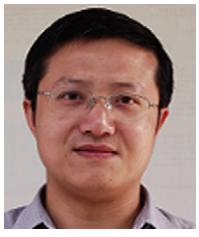
References

[1] Fuzzgrind an automatic fuzzing tool Fuzzgrind. http://esec-lab.sogeti.com/dotclear/index.php?pages/Fuzzgrind.
[2] P. Godefroid, M. Levin, D. Molnar, Automated whitebox fuzz testing, in: Proceedings of the Network and Distributed System Security Symposium, 2008.
[3] J.C. King, Symbolic execution and program testing, Journal of the ACM 19 (7) (1976) 385–394.
[4] G.J. Myers, The Art of Software Testing, Wiley, 1979.
[5] E.J. Schwartz, T. Avgerinos, D. Brumley, All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), in: Proceedings of IEEE Symposium on Security and Privacy, 2010, pp. 317-331.
[6] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, D. Song, A symbolic execution framework for JavaScript, in: Proceedings of IEEE Symposium on Security and Privacy, 2010, pp. 513–528.
[7] T.L. Wang, T. Wei, G.F. Gu, W. Zou, TaintScope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection, in: Proceedings of IEEE Symposium on Security and Privacy, 2010, pp. 497–512.
[8] C. Cadar, D. Dunbar, D. Engler, Klee: unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the USENIX Symposium on Operating System Design and Implementation, 2008.
[9] C. Cadar, D. Engler, Execution generated test cases: how to make systems code crash itself, Lecture Notes in Computer Science 3639 (2005) 2–23.
[10] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, D.R. Engler, EXE: a system for automatically generating inputs of death using symbolic execution, in: Proceedings of the ACM Conference on Computer and Communications Security, 2006.
[11] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, 2005, pp. 213–223.
[12] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for *C*, in: Proceedings of the Joint 10th European Software Engineering Conference, ESEC, and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE-13, 2005, pp. 263–272.
[13] J. Newsome, B. Karp, D. Song, Polygraph: automatically generating signatures for polymorphic worms, in: Proceedings of IEEE Symposium on Security and Privacy, 2005, pp. 226–241.
[14] Z.C. Li, M. Sanghi, Y. Chen, M.Y. Kao, B. Chavez, Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience, in: Proceedings of IEEE Symposium on Security and Privacy, 2006, pp. 32–46.
[15] X.S. Zhang, T. Chen, D.P. Chen, Z. Liu, SISG: self-immune automated signature generation for polymorphic worms, COMPEL—The International Journal for Computation and Mathematics in Electrical and Electronic Engineering 29 (2) (2010) 445–467.
[16] D. Brumley, J. Newsome, D. Song, H. Wang, S. Jha, Towards automatic generation of vulnerability-based signatures, in: Proceedings of the IEEE Symposium on Security and Privacy, 2006, pp. 2–16.
[17] D. Brumley, J. Newsome, D. Song, H. Wang, S. Jha, Theory and techniques for automatic generation of vulnerability-based signatures, IEEE Transactions on Dependable and Secure Computing 5 (4) (2008) 224–241.
[18] D. Brumley, H. Wang, S. Jha, D. Song, Creating vulnerability signatures using weakest preconditions, in: Proceedings of the IEEE Computer Security Foundations Symposium, 2007, pp. 311–325.
[19] Z.K. Liang, R. Sekar, Fast and automated generation of attack signatures: a basis for building self-protecting servers, in: Proceedings of the ACM Conference on Computer and Communications Security, 2005, pp. 213–222.
[20] J. Newsome, D. Brumley, D. Song, J. Chamcham, X. Kovah, Vulnerability-specific execution filtering for exploit prevention on commodity software, in: Proceedings of the Network and Distributed System Security Symposium, 2006.
[21] D. Brumley, C. Hartwig, M.G. Kang, Z.K. Liang, J. Newsome, P. Poosankam, D. Song, Bitscope: automatically dissecting malicious binaries, Technical Report CS-07-133, 2007.
[22] D. Brumley, C. Hartwig, Z.K. Liang, J. Newsome, P. Poosankam, D. Song, H. Yin, Automatically identifying trigger-based behavior in malware, in: Botnet Detection: Countering the Largest Security Threat, in: Advances in Information Security, vol. 36, Springer-Verlag, 2008.
[23] A. Moser, C. Kruegel, E. Kirda, Exploring multiple execution paths for Malware analysis, in: Proceedings of IEEE Symposium on Security and Privacy, 2007, pp. 229–243.
[24] D. Song, D. Brumley, Yin Heng, J. Caballero, I. Jager, G.K. Min, Z.K. Liang, J. Newsome, P. Poosankam, P. Saxena, Bitblaze: a new approach to computer security via binary analysis, in: Proceedings 4th International Conference, 2008, pp. 1–25.
[25] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: Proceedings of the IBM Workshop on Logics of Programs, vol. 131, 1981, pp. 52–71.
[26] S. Cha, J. Yoo, A safety-focused verification using software fault trees, Future Generation Computer Systems (2011).
[27] G. Fraser, F. Wotawa, P.E. Ammann, Testing with model checkers: a survey, Software Testing Verification and Reliability 19 (3) (2009) 215–261.
[28] A. Biere, A. Climatti, E.M. Clarke, Y.S. Zhu, Symbolic model checking without BDDs, in: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, 1999, pp. 193–207.
[29] P. McMinn, Search-based software test data generation: a survey, Software Testing Verification and Reliability 14 (2) (2004) 105–156.
[30] M. Harman, Automated test data generation using search based software engineering, in: Proceedings of International Conference on Software Engineering, 2007.
[31] M. Harman, R. Hierons, M. Proctor, A new representation and crossover operator for search-based optimization of software modularization, in: Proceedings of the 2002 Conference on Genetic and Evolutionary Computation, GECCO'02, 2002, pp. 1351–1358.
[32] P. Baker, M. Harman, K. Steinhofel, A. Skaliotis, Search based approaches to component selection and prioritization for the next release problem, in: Proceedings of the 22nd IEEE International Conference on Software Maintenance, ICSM'06, 2006, pp. 176–185.
[33] S. Bouktif, H. Sahraoui, G. Antoniol, Simulated annealing for improving software quality prediction, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, GECCO'06, 2006, pp. 1893–1900.
[34] C. Gil, J. Ortega, A.F. Diaz, M.D.G. Montoya, Annealing-based heuristics and genetic algorithms for circuit partitioning in parallel test generation, Future Generation Computer Systems 14 (5–6) (1998) 439–451.
[35] P.M.S. Bueno, M. Jino, Automatic test data generation for program paths using genetic algorithms, International Journal of Software Engineering and Knowledge Engineering 12 (6) (2002) 691–709.
[36] X.B. Wang, N. Su, Automatic test data generation for path testing using genetic algorithms, in: Proceedings of 3rd International Conference on Measuring Technology and Mechatronics Automation, vol. 1, 2011, pp. 596–599.
[37] P. Joshi, K. Sen, M. Shlimovich, Predictive testing: amplifying the effectiveness of software testing, Technical Report No. UCB/EECS-2007-35, 2007.
[38] N. Tillmann, J. De Halleux, Pex-white box test generation for.NET, in: Proceedings of 2nd International Conference on Tests and Proofs, 2008, pp. 134–153.
[39] P. Godefroid, Compositional dynamic test generation, in: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2007, pp. 47–54.
[40] S. Anand, P. Godefroid, N. Tillmann, Demand-driven compositional symbolic execution, in: Proceedings of 14th International Conference on Theory and Practice of Software, 2008, pp. 367–381.
[41] P. Godefroid, A.V. Nori, S.K. Rajamani, S.D. Tetali, Compositional may-must program analysis: unleashing the power of alternation, in: Proceedings of the Annual ACM Symposium on Principles of Programming Languages, 2010, pp. 43–55.
[42] P. Godefroid, D. Luchaup, Automatic partial loop summarization in dynamic test generation, in: Proceedings of 2011 International Symposium on Software Testing and Analysis, 2011, pp. 23–33.
[43] P. Godefroid, A. Kiezun, M.Y. Levin, Grammar-based whitebox fuzzing, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2008, pp. 206–215.
[44] R. Majumdar, R.G. Xu, Directed test generation using symbolic grammars, in: Proceedings of ACM/IEEE International Conference on Automated Software Engineering, 2007, pp. 134–143.
[45] R.G. Xu, P. Godefroid, R. Majumdar, Testing for buffer overflows with length abstraction, in: Proceedings of the International Symposium on Software Testing and Analysis, 2008, pp. 27–37.
[46] R. Majumdar, R.G. Xu, Reducing test inputs using information partitions, in: Lecture Notes in Computer Science, vol. 5643, 2009, pp. 555–569.
[47] P. Godefroid, J. Kinder, Proving memory safety of floating-point computations by combining static and dynamic program analysis, in: Proceedings of the International Symposium on Software Testing and Analysis, 2010, pp. 1–11.
[48] B. Elkarablieh, P. Godefroid, M.Y. Levin, Precise pointer reasoning for dynamic test generation, in: Proceedings of International Conference on Software Testing and Analysis, 2009, pp. 129–140.
[49] M.I. Emmi, R. Majumdar, K. Sen, Dynamic test input generation for database applications, in: Proceedings of ACM International Symposium on Software Testing and Analysis, 2007, pp. 151–162.
[50] K. Taneja, Y. Zhang, T. Xie, MODA: automated test generation for database applications via mock objects, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2010, pp. 289–292.
[51] P. Godefroid, Higher-order test generation, in: The Proceedings of 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011, pp. 258–269.
[52] A.K. Gupta, T.A. Henzinger, R. Majumdar, A. Rybalchenko, R.G. Xu, Proving non-termination, SIGPLAN Notices 43 (1) (2008) 147–158.
[53] G.C. Necula, S. McPeak, S.P. Rahul, W. Weimer, CIL: intermediate language and tools for analysis and transformation of *C* programs, in: Proceedings of Conference on Compiler Construction, 2002, pp. 213–228.
[54] CIL, C intermediate language: http://cil.sourceforge.net/.

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006

[55] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, Vijay Janapa Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005, pp. 190–200.
[56] Pin—A dynamic binary instrumentation tool: http://www.pintool.org/.
[57] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, SIGPLAN Notices 42 (6) (2007) 89–100.
[58] Valgrind: http://valgrind.org/.
[59] Coverity. http://www.coverity.com/.
[60] AppScan. http://www-01.ibm.com/software/awdtools/appscan/.
[61] WebInspect. www.hp.com/go/appsec.
[62] P.J. Schroeder, B. Korel, Black-box test reduction using input–output analysis, in: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, 2000, pp. 173–177.
[63] L.H. Tahat, B. Vaysburg, B. Korel, A.J. Bader, Requirement-based automated black-box test generation, in: Proceedings of IEEE Computer Society's International Computer Software and Applications Conference, 2001, pp. 489–495.
[64] H. Kim, Y. Choi, D. Lee, D. Lee, Practical security testing using file fuzzing, in: International Conference on Advanced Communication Technology, vol. 2, 2008, pp. 1304–1307.
[65] H. Kim, Y. Choi, D. Lee, Efficient file fuzz testing using automated analysis of binary file format, Journal of Systems Architecture (2010).
[66] C.S. Pasareanu, W. Visser, A survey of new trends in symbolic execution for software testing and analysis, International Journal on Software Tools for Technology Transfer 11 (4) (2009) 339–353.
[67] D. Coppit, J. Lian, Yagg: an easy-to-use generator forstructured test inputs, in: Conference of Automated Software Engineering, 2005, pp. 356–359.
[68] R. Lammel, W. Schulte, Controllable combinatorial coverage in grammar-based testing, in: Conference of Testing of Communicating Systems, 2006.
[69] D.M. Leonardo, B. Nikolaj, Z3: an efficient SMT solver, in: Lecture Notes in Computer Science, vol. 4963, 2008, pp. 337–340.
[70] STP, Simple theorem prover: http://sourceforge.net/projects/stp-fast-prover.
[71] CVC3: http://www.cs.nyu.edu/acsys/cvc3/.
[72] Yices: an SMT solver. http://yices.csl.sri.com/.
[73] P. Godefroid, M.Y. Levin, D. Molnar, Active property checking, in: Proceedings of the 7th ACM International Conference on Embedded Software, 2007, pp. 207–216.
[74] V. Chipounov, V. Georgescu, C. Zamfir, G. Candea, Selective symbolic execution, in: Proceeding of 5th Workshop on Hot Topics in System Dependability, HotDep, 2009.
[75] V. Chipounov, V. Kuznetsov, G. Candea, S2E: a platform for in-vivo multi-path analysis of software systems, Computer Architecture News 39 (1) (2011) 265–278.
[76] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, 1979.
[77] I. Erete, A. Orso, Optimizing constraint solving to better support symbolic execution, in: Proceedings of 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, 2011, pp. 310–315.
[78] E. Uzuncaova, D. Garcia, S. Khurshid, D. Batory, Testing software product lines using incremental test generation, in: Proceedings of International Symposium on Software Reliability Engineering, 2008, pp. 249–258.
[79] E. Uzuncaova, S. Khurshid, D. Batory, Incremental test generation for software product lines, IEEE Transactions on Software Engineering 36 (3) (2010) 309–322.
[80] P. Saxena, P. Poosankam, S. McCamant, D. Song, Loop-extended symbolic execution on binary programs, in: Proceeding of 8th International Symposium on Software Testing and Analysis, 2009, pp. 225–236.
[81] S. Bhansali, W. Chen, S. De Jong, A. Edwards, M. Drinic, Framework for instruction-level tracing and analysis of programs, in: Second International Conference on Virtual Execution Environments, 2006.
[82] Y. Hamadi, Disolver: a distributed constraint solver, Technical Report MSR-TR-2003-91, Microsoft Research, 2003.
[83] Microsoft. Net framework general reference—profiling (unmanaged api reference). http://msdn2.microsoft.com/en-us/library/ms404386.aspx.
[84] C. Lattner, V. Adve, LLVM: a compilation framework for lifelong program analysis & transformation, in: Proceedings of the International Symposium on Code Generation and Optimization, 2004.
[85] LP solve. http://lpsolver.sourceforge.net/5.5/.
[86] D.A. Molnar, D. Wagner, Catchconv: symbolic execution and run-time type inference for integer conversion errors, Technical Report, University of California, Berkeley, 2007-23, 2007.
[87] F. Bellard, QEMU, a fast and portable dynamic translator, in: Proceedings of the USENIX Annual Technical Conference, 2005.
[88] P. Godefroid, S.K. Lahiri, C.R. González, Incremental compositional dynamic test generation, Technique Report MSR-TR-2010-11, 2010.
[89] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounuv, G. Candea, Cloud9: a software testing service, Operating Systems Review 43 (4) (2009) 5–10.
[90] G. Candea, S. Bucur, C. Zamfir, Automated software testing as a service, in: Proceedings of the 1st ACM Symposium on Cloud Computing, 2010, pp. 155–160.
[91] G.D. Li, I. Ghosh, S.P. Rajan, KLOVER: a symbolic execution and automatic test generation tool for C + + programs, in: Proceedings of 23rd International Conference on Computer Aided Verification, 2011.
[92] T. Xie, N. Tillmann, J. de Halleux, W. Schulte, Fitness-guided path exploration in dynamic symbolic execution, in: Proceedings of IEEE/IFIP International Conference on Dependable Systems & Networks, DSN, 2009, pp. 359–368.
[93] C. Csallner, N. Tillmann, Y. Smaragdakis, DySy: dynamic symbolic execution for invariant inference, in: Proceedings of ACM/IEEE 30th International Conference on Software Engineering, 2008, pp. 281–290.
[94] R. Majumdar, K. Sen, Hybrid concolic testing, in: Proceedings of International Conference on Software Engineering, 2007, pp. 416–425.
[95] P. Boonstoppel, C. Cadar, D. Engler, RWset: attacking path explosion in constraint-based test generation, in: Proceedings of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008, pp. 351–366.
[96] SMT-COMP 2010. http://www.smtcomp.org/2010/.
[97] S. Khurshid, C.S. Pasareanu, V. Willem, Generalized symbolic execution for model checking and testing, in: Proceedings of 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2003, pp. 553–568.
[98] A. Griesmayer, B. Aichernig, E.B. Johnsen, R. Schlatte, Dynamic symbolic execution of distributed concurrent objects, in: Proceedings of International Conference on Formal Techniques for Distributed Systems and 29th IFIP WG 6.1, 2009, pp. 225–230.
[99] S. Bumler, M. Balser, F. Nafz, W. Reif, G. Schellhorn, Interactive verification of concurrent systems using symbolic execution, AI Communications 23 (2–3) (2010) 285–307.
[100] S.F. Siegel, A. Mironova, G.S. Avrunin, L.A. Clarke, Combining symbolic execution with model checking to verify parallel numerical programs, ACM Transactions on Software Engineering and Methodology 17 (2) (2008) 1–34.
[101] S. Anand, C.S. Pasareanu, W. Visser, JPF-SE: a symbolic execution extension to Java PathFinder, in: Proceedings of 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2007, pp. 134–138.
[102] K. Lakhotia, N. Tillmann, M. Harman, J. de Halleux, FloPSy—search-based floating point constraint solving for symbolic execution, in: Proceedings of 22nd IFIP WG6.1 International Conference on Testing Software and Systems, 2010, pp. 142–157.
[103] D. Singer, A. Monnet, JaCK-SAT: a new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check, in: Proceedings of 7th International Conference on Parallel Processing and Applied Mathematics, 2008, pp. 249–258.
[104] G.J. Holzmann, D. Bosnacki, Multi-core model checking with SPIN, in: Proceedings of IEEE International Parallel and Distributed Processing Symposium, 2007, pp. 1–8.
[105] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, D. Marinov, Parallel test generation and execution with Korat, in: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT on the Foundation of Software Engineering, 2007, pp. 135–144.
[106] P. Godefroid, D. Molnar, Fuzzing in the cloud (position statement), Technique Report MSR-TR-2010-29, 2010.
[107] Y.D. Wang, NVIDIA CUDA architecture-based parallel incomplete SAT solver, Master Project Final Report, Rochester Institute of Technology, 2010.

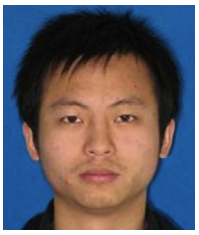
Ting Chen received the BS degree in mathematics from University of Electronic and Technology of China (UESTC), Chengdu, in 2007, and the MS degree in computer science from UESTC in 2010. Now he is pursuing his Ph.D. degree at UESTC. He has published several articles in prestigious journals and conferences in the fields of malware analysis, intrusion detection and software testing. He has obtained several Chinese patents with regard to computer security. His current research includes software reliability, software test case generation and software verification.



Xiao-song Zhang graduated with a BS degree in dynamics engineering from Shanghai Jiaotong University, Shanghai, in 1990, and an MS and Ph.D. degree in computer science from UESTC, Chengdu, in 2011. Now he is a Professor in computer science at UESTC. He has worked on numerous projects in both research and development roles. These projects include device security, intrusion detection, malware analysis, software testing and software verification. He has coauthored a number of research papers on computer security. His current research involves in software reliability, software vulnerability discovering, software test case generation and reverse engineering.

Shi-ze Guo is a Professor in Beijing University of Posts and Telecommunications. He has served as research scientist at Institute of North Electronic Equipment. His current research is relevant to software engineering and formal methods.

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006



Hong-yuan Li received the BS degree in computer science from UESTC, Chengdu, in 2010. Now he is pursuing his MS degree in computer science at UESTC. His current research includes malware detection on smart phone and software reliability analysis.

Yue Wu is a Professor in UESETC. He had severed as Dean of School of Computer Science and Engineering in UESTC from 2001 to 2006. He was the Dean of School of Information and Software Engineering in UESTC from 2002 to 2006. He worked as the Dean of School of Software in Chengdu College of UESTC from 2002 to 2004. He is a committee member of several journals including *Journal of UESTC, Journal of Computer Applications, Software World*. He has presided over a number of international conferences such as *CIDE 2005* and *ISNN 2006*. He has published more than 70 research papers and authored 3 books. His current research focuses on grid computing, database systems and data mining.

Please cite this article in press as: T. Chen, et al., State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems (2012), doi:10.1016/j.future.2012.02.006