Static Analysis and Symbolic Code Execution

Dhiren Patel, Madhura Parikh and Reema Patel*

* National Institute of Technology, Surat, Gujarat, India
E-mail: (dhiren29p, madhuraparikh and reema.mtech)@gmail.com

As software becomes increasingly pervasive and affects critical areas of application, verifying the correctness of programs can no longer be neglected. Several approaches in the past have utilized *testing* to prove program correctness, but this is an incomplete approach. The second alternative is to perform *static analysis and model checking*. While this is an exhaustive approach, it has several limitations viz; high cost, poor scalability, spurious warnings. In this paper, we explore Symbolic execution that takes the middle way between these two extremes, and has the potential to be applicable in real world settings. Initially proposed in the 1970's, symbolic execution has recently gained focus amongst researchers. Starting from its birth, we survey tools that have successfully implemented symbolic execution and the modifications that have been proposed to make it widely applicable. We also examine the research challenges that exist with this approach and how well-received it has been in industry.

Index Terms : Static Analysis, Symbolic Execution, Dynamic Test Generation, Concolic Execution, Compositional Testing.

1. Introduction

As software is being used in increasingly complex and critical applications, the importance of program verification cannot be stressed enough. Conventionally two approaches to verification have been cited - static analysis and dynamic analysis. Both these approaches however, have their own short-comings that have curtailed their use in practice. We first introduce these conventional methods of verification, examining their pros and cons. We then look at the idea of symbolic execution, as it was originally proposed by King [1] in 1976. Symbolic execution was essentially a static analysis method. However it has recently evolved to a new form, imbibing advantages of both static and dynamic code analysis. The possible areas of application of this novel form of symbolic execution are then discussed in later sections.

1.1 Static Analysis of Programs

An excellent introduction and review of static analysis can be found in the book by Nielson [2]. Static analysis of code, as the name suggests tries to analyze the program behavior without actually executing the code. Thus the correctness of the program is analyzed by applying formal techniques on the source code or the object code. One of the earliest examples of a static analysis tool is the wellknown Lint program for C. Such bug finding tools generally use static analysis only in a shallow way, meaning that they look out for unexpected constructs in the program. This means that they will also raise an increased number of false alarms, while missing out on subtle errors. Some important concepts associated with any static analysis approach are whether the approach is sound and complete. A sound approach is guaranteed to find any true error in the program; a complete one will find only errors that are true. Some of the commonly used methods of formal static analysis are summarized below:

1.1.1 Type Systems

Types are most widely used in static analysis. This is because most of the major programming languages use the concept of types. A type is just a set of values. For e.g the type Int would denote all the integers whereas the type Bool in a language like C ++ would represent the set true, false. The concept of types in fact follows from the concept of abstract interpretation, which is used in any static analysis approach. Ideally static analysis should be able to represent all possible executions of a program. This is however un-decidable in practice. So a compromise is to consider only abstract values rather than the infinite set of all possible concrete values. Thus *Int* is an abstraction for the infinite possible integers.

1.1.2 Data Flow Analysis

Data flow analysis usually requires us to construct a Control Flow Graph (CFG). This is a directed graph in which each node is a statement and edges represent the flow of control. Data flow analysis tries to find facts about the program by considering all possible states of the program. It includes different analyses such as liveliness analysis, reaching definitions, etc., that are used in optimizing compilers. Detailed literature on data-flow analysis may be

found in any standard text on compilers such as [3].

1.1.3 Model Checking

Model checking usually uses some form of temporal logic such as finite automata, to represent the specifications of a program. The state space is then exhaustively searched, to ensure that the specifications are met correctly. It is an approach that suffers from state space explosion. Bounded model checking examines only a prefix of all possible executions. However this may result in later errors being missed.

1.2 Dynamic Analysis of Programs

Dynamic analysis is mainly done through testing and debugging. Testing tries to run a program on a subset of the possible inputs to observe if a failure occurs or not. Since it is not possible to run the program exhaustively over all possible inputs, testing has the problem of absence of coverage. One important concern is that the test case generation should be automated if it is to be applicable to commercial programs that are thousands of lines long. The way these tests are generated is important, since naively generating the test cases in any random manner may likely miss out on several errors.

Debugging involves pinpointing the exact location of errors, and fixing these up. In earlier times, debugging was usually achieved by inserting print statements at suspicious locations in the program. With the advent of IDEs and better debuggers, this situation is somewhat eased, however debugging even today is highly arcane and tedious, requiring much manual involvement.

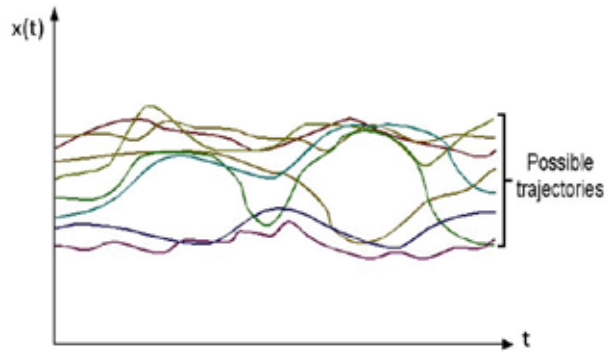

Fig. 1 : All the trajectories in a program execution space [4]

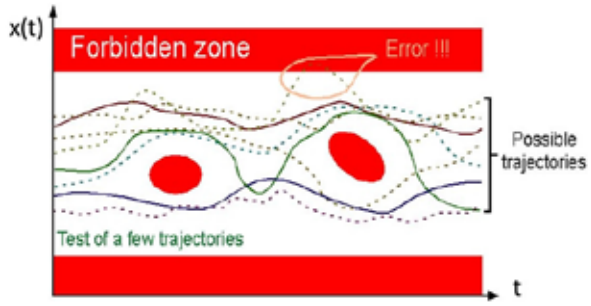

Fig. 2 : Trajectory coverage with model-checking [4]

A good visualization of these different methods is shown in Figures 1 2 3. In each of these figures x(t) is a vector of the input state and output of the program that evolves as a function of time t. Figure 1 shows the possible different trajectories that a program may take up when executed. As Figure 2 shows, model checking tries to cover each of the trajectories, however to be practical, it must be bounded. Thus it misses out on any late errors. On the other hand testing as shown in Figure 3 will leave out some of the trajectories entirely while some are covered completely. Clearly even this is not satisfactory.

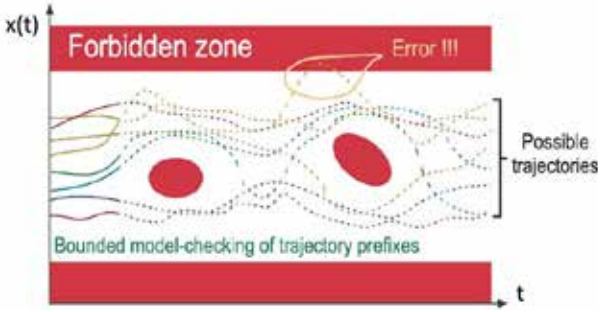

Fig. 3 : Trajectory coverage with dynamic analysis [4]

1.3 Symbolic Execution : Old Paradigm Gets New Life

The concept of symbolic execution was first proposed by King in his seminal paper [1]. We first try to understand the concept in the form it was originally proposed. As we have already noted, program verification conventionally is done either through proving by formal techniques, or by testing with a random input set. Symbolic execution is a middle way between these two extremes. In symbolic execution, the program is executed, but the input fed to the program is symbolic rather than concrete. This means that each symbolic input stands for an entire class of inputs rather than individual input. Thus while testing tries to check the program only on a very small subset of the possible inputs and verification checks the program over all of the possible inputs, symbolic execution checks the program over classes of the input. So it may be considered to be a more generalized form of testing or a less stringent verification.

Whenever a program is executed concretely, the program state usually consists of the program counter and the values of the program variables. When a program is to be executed symbolically, one additional piece of information must also be maintained in the so-called symbolic program state. This is the path condition (pc). The pc is the accumulation of all the criteria that the input must satisfy for an execution to follow the associated path [1]. As we encounter, conditions or branches in the program flow, such as IF statements, these conditions are conjoined with the pc along that path. Thus a pc exists, corresponding to each alternative path that the program may follow. This pc is thus a Boolean formula. In order to reason about whether a particular control flow is possible for the program, we should check whether the corresponding Boolean formula that represents its pc is satisfiable. For this powerful SMT/SAT solvers are used. Of course while many satisfiability problems are beyond

the scope of these solvers, many solvers do exist for linear equations, e.g Z3, STP, Yices, etc.

Thus corresponding to each program, we can generate the execution tree that shows all the possible execution paths of the program. One such symbolic execution tree and the corresponding code snippet appear in the Figure 4. Here it is seen that at each point in a particular path, the pc is updated with some additional constraints if necessary. If the final pc for a particular path is found to be un-satisfiable, then that state of the program is unreachable. The various path conditions accrued may be used to generate test cases. All concrete values that satisfy the path condition for a particular path are guaranteed to follow that path when inputted to the program.

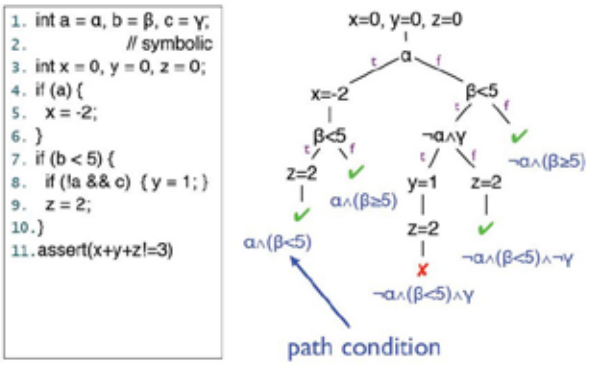

Fig. 4 : A symbolic execution tree [5]

It is important to note that classical symbolic execution, will work only for sequential programs. In recent times, several extensions to this classical approach have been proposed. Excellent reviews of the current trends in symbolic execution are available in [6] [7]. These new approaches include ideas like generalized symbolic execution, concolic symbolic execution, compositional symbolic execution, etc. They have made it possible to extend symbolic execution to multi-threaded programs and advanced programming constructs, such as recursive data-structures. We shall survey a few of these ideas in a later section.

1.4 Applications of Symbolic Execution

The paper by Visser et al [8] covers a depth of areas to which symbolic execution may be applied. Here we enlist some major areas of focus.

1.4.1 Test case generation

This is one of the traditional uses of symbolic execution. Since symbolic execution deals with classes of input, it can be used to automate generation of test-cases with a high degree of coverage. This has recently become well known as whitebox testing.

1.4.2 Proof of program properties

Symbolic execution can be successfully applied for proving that certain assertions, say for instance loop invariants, are indeed maintained when a program is executed.

1.4.3 Static detection of run time errors

This uses symbolic execution to detect if program states, that may lead to run time violations, exist. Such analysis is especially useful in the case of malware analysis, where it may be expensive to run the code in order to observe its noxious behavior.

1.4.4 Invariant inference

By using symbolic execution, we may predict the pre or post conditions that are likely to be maintained by the program.

1.4.5 Parallel numerical program analysis

This is a new application of symbolic execution to establish that the parallel program is indeed equivalent to its corresponding sequential counterpart.

1.4.6 Differential symbolic execution

This is again an emerging area. Here two programs are compared to find out the logical difference between them. It may be used in software development and maintenance to ensure for instance that re-factored code is equivalent to the original source.

The rest of the paper is organized as follows: In section 2 we discuss some tools such as KLEE that are developed using symbolic execution. Some exciting new trends that are emerging in symbolic execution are described in section 3. Section 4 examines some of the major challenges that symbolic execution techniques must counter. Finally, in section 5, we see how well accepted the various symbolic execution tools have been in industry.

2. Symbolic Execution: Case Studies

In this section we explore two tools that have successfully used symbolic execution for program analysis. Several breakthrough tools have been proposed, but we focus on only two such contemporary tools. Our first case study is on KLEE - which is an automatic test case generation framework for C. Since C is a conventional statically typed language, the study of KLEE will acquaint us with how symbolic execution can be applied in such a mainstream language. Our second case study deals with Kudzu - which is an automated vulnerability analysis tool for JavaScript. A study of Kudzu will give us a flavour of how symbolic execution may be applied to solve string constraints, a major application challenge.

2.1 Case Study 1: KLEE

KLEE is one of the most successful implementations of symbolic execution as a means for automated generation of high coverage test cases. The paper that provides detailed technical overview of KLEE is [9]. KLEE has two major execution goals:

1) Try to hit every line of code. This will help achieve high coverage.
2) At each dangerous operation that is detected, try to

generate the test case that could potentially cause the error. Do this by solving that path's path condition.

The authors of this paper were previously involved in the development of a similar tool EXE [10]. They have based KLEE on their previous experiences and have shown that KLEE generated tests could indeed achieve high test coverage of over 90%, sometimes even beating manually developed test suites. Here we briefly explore some major features.

2.1.1 Architecture

The KLEE framework is designed for:

Simplicity: It operates on the well known LLVM byte code. This means that it can be applied to a number of languages that are supported by LLVM. LLVM has a RISC like instruction set and the byte code is generated from the source code, which is compiled using the well known GCC compiler. Thus no special modifications need to be made in order to have the code analyzed by KLEE. KLEE can take in the raw code directly and generate the test cases, and is therefore quick and easy to use.

Scalability: Since there can be an exponential increase in the number of states, KLEE uses compact state representation. It borrows the well-known principle of copy on write (vfork vs. fork) from Linux, to reduce memory requirements. It also implements the heap as immutable and shares the heap amongst multiple states. Finally it uses compact expression representation (e.g. when it has expression x+1=10 simply store this as the expression x=9.)

Speed: It achieves speed by trying to minimize the calls to the SAT solver, since these are the most expensive. It also uses simplified expressions as input to the constraint solver, and uses the concept of caches to dramatically improve speed.

The figure 5 shows how the different components of KLEE integrate to produce very accurate test cases for the sample code snippet on the lower-left in the figure.

2.1.2 Solver Optimization

The major time consumed by most symbolic execution tools is in trying to solve the constraints. Thus KLEE has used some clever techniques to greatly reduce this time cost. It uses the concept of constraint independence to give only relevant portions of the memory to the solver for a particular constraint rather than the complete store of all the variables. It uses the concept of *counter-example cache* to store results of previous queries. These results are checked to see if they satisfy a current query, since it is much easier to check the solution rather than solve. Also it tries to see if the cached solution has some subset, superset relation to the current constraint's solution state.

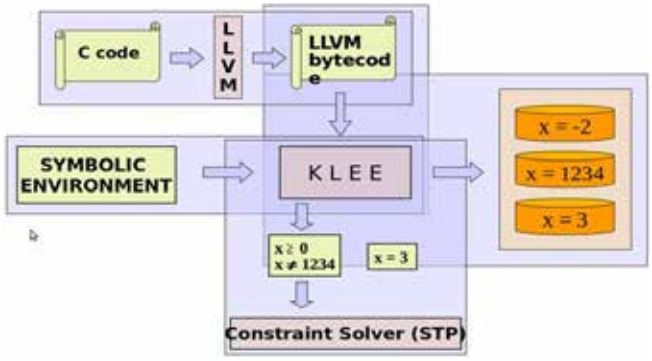

Fig. 5 : Different components of KLEE [11]

2.1.3 Major Challenges

KLEE has tried to answer two challenges:

1) How to model the interaction of the program with the environment: Many programs have command line arguments, environment variables, file handling, etc. To provide a realistic environment in such cases, KLEE makers have tried to provide a symbolic environment, which the program sees when it is executed symbolically. This tries to model its interaction with real-life system calls and OS resources.
2) How to address exponential paths through the code: The state space may grow exponentially. So KLEE uses two strategies to determine which state to choose next from the current state. The first strategy is *Random Path Selection* - this uses a binary tree to represent active states, and the tree is randomly traversed from the root. The approach is biased towards states higher up in the tree where simpler constraints need to be solved. The second is *Coverage Optimized Search* where it uses some heuristics to compute weights for each state and then randomly selects the state that is likely to offer better coverage.

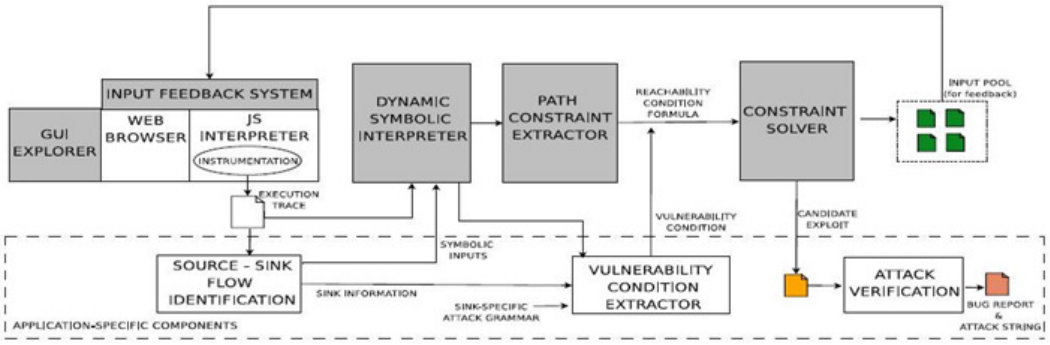

Fig. 6 : The various components in the design of Kudzu [12]

2.1.4 Evaluation Highlights

The metric for evaluation is line coverage. KLEE generated test cases for Coreutils, Minix, Busybox, etc. It was able to outperform manually designed test suites developed over a period of 15 years. The coverage was over 90%. It found several major flaws in GNU Coreutils, which is one of the most heavily tested program suite. Thus it was shown to have great potential for real world application.

2.2 Case Study 2: Kudzu

Kudzu [13] is an automated tool for finding client side code injection vulnerabilities in JavaScript. It is the first attempt of this kind. In a language like JavaScript, the input is in the form of strings. This means that the solver should be able to solve string constraints. To enable this, Kudzu introduces an expressive constraint language and a constraint solver Kaluza specifically geared for handling string constraints, rather than numeric ones.

2.2.1 System Design

For the rich web applications that are created using AJAX, the input space is conceptually divided into 2: event space - that deals with various event handler code e.g. mouse clicks or form submissions, that may occur in any order - and secondly value space - these are the values provided by the user in form fields, text areas, URL parameters, etc. Kudzu uses the concept of GUI exploration to handle vulnerabilities associated with the event space while using symbolic execution to handle the untrusted input. Next, it incorporates the powerful string constraint solver Kaluza. For this, the original JavaScript instructions are translated to a simple language - JASIL. A block diagram for the system architecture appears in Figure 2.1.1. Kudzu is the automated tool for finding security vulnerabilities in JavaScript code.

2.2.2 Constraint Language

The constraint language is very expressive for string constraints. Various constraints are provided to check if a given string matches a given regular expression, comparing two strings for equality or concatenation, comparing the length of two strings and checking the length of a string against some integer. All these facilities make the constraint solver more powerful than its contemporaries. Moreover in-spite of its high expressiveness, the constraint language is simple, making the constraint solver efficient.

2.2.3 Constraint Solver

Kaluza is a SAT based SMT constraint solver. It first of all takes in as input the JavaScript code and translates it to the core constraint language using a DFA-based approach. Next, it solves for various constraints such as length and integer constraints, etc. Finally it takes the input strings and translates them into bit vector notation, by concatenating the binary representation of consecutive characters in the string. It then checks to see if the bit vector notation of the string contents satisfies the constraints using the SMT solver. It also uses the concept of k-boundedness, and expects the user to input the value of k- the maximum length of the strings to be included in the search space. This is essential as otherwise the problem would become unbounded and unsolvable practically.

2.2.4 Evaluation

The solver is very successful for detecting client side code injection attacks. It was able to detect 2 new vulnerabilities when tested on 18 popular web apps and also 9 known vulnerabilities that had earlier been detected using manual testing. The constraint solver is also highly efficient, when a constraint is satisfiable, it can find the solution in under a second, when it is not it may take up-to 50s to report failure. The only requirement is that the user must provide an upper bound on the search space.

3. Exciting New Areas in Symbolic Execution Research

Several recent trends have emerged that attempt to carry out some form of hybrid analysis that tries to combine symbolic execution with other techniques in an attempt to improve the overall performance. Several such techniques have been proposed for testing critical applications such as in NASA [14] [15]. We survey some such trends in the present section.

3.1 Concolic Execution: Combining Symbolic and Concrete Analysis

A very concise tutorial on this technique is available in [16]. Concolic execution combines both random testing as well as symbolic execution. Whenever a concrete execution is performed using random inputs, the path conditions for that particular path are also simultaneously constructed. In the next run the gathered constraints are used to generate new inputs, which will drive the execution along a different branch. For e.g. by negating one constraint at a branch point, the new input generated will then guide the program along the other branch. This is repeated until no new constraints can be generated. Thus significant coverage is attained. At the same time, if some constraints are too difficult for the solver, then the concrete input comes to rescue, by replacing some of the symbolic variables with concrete values. A good case study of such a hybrid system is DART [17]. DART is an algorithm for dynamic test case generation using random testing and symbolic execution. It is one of the precedents in this area, originally developed at Bell Labs. Here the major advantages of this approach are that the random test generation is fully automated, not requiring any manual intervention, thanks to the path constraints generated by symbolic execution. Unlike random testing it is also more effective, since the program execution may be directed along specific paths by using the path constraints. It also tackles the difficulties of pure

symbolic execution where constraints may be too difficult to solve. Table 1 shows some recent tools that have been used symbolic execution.

3.2 Compositional Symbolic Execution

Compositional symbolic execution is an attempt to extend concolic execution to make it more scalable by intelligently avoiding state space explosion. The seminal paper introducing this concept is [18]. Here the major concept used is to generate summaries for individual functions. These include information such as the function pre and post conditions, etc. These summaries are generated in top down manner in context of the caller-callee relationships between the functions. Thus if a function f() calls a function g(), then g()'s summary is generated first and used later when the function f() is being analyzed. By using such an approach, the overall complexity would be lesser than if both the functions are analyzed together. With this new approach, the complexity may be curbed while at the same time not sacrificing the code coverage.

For this approach, the authors of [18] have proposed an algorithm called SMART. They have reported that SMART is able to achieve the same level of coverage as the earlier DART algorithm. The Fig. 7 shows an experimental comparison of the number of runs with SMART and DART. SMART shows only a linear growth as compared to DART, which may show exponential scaling in the number of possible execution paths. The major challenge is to generate the summaries intelligently, rather than naively generating all possible summaries, since we want to counter path explosion.

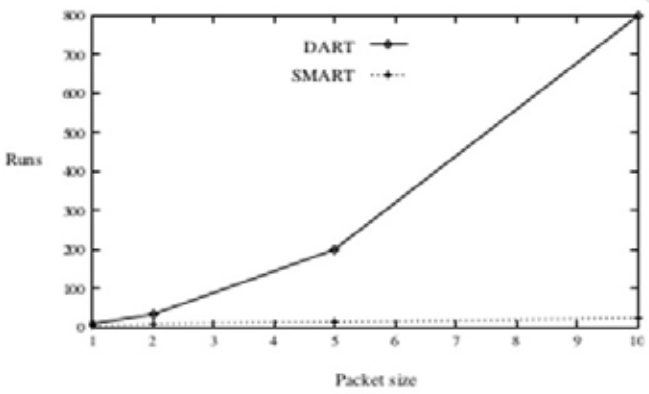

Fig. 7 : Experimental comparison of the number of runs with SMART and DART [18]

3.3 Automated Whitebox Fuzz Testing

Fuzz testing is a heavily used random testing method for detecting security vulnerabilities. This basically involves sending input data to the application and then randomly mutating the input and then testing the results. It is a form of black box testing. Whitebox fuzz testing [19] is an attempt to combine fuzz testing with symbolic execution, in an attempt to facilitate dynamic and automated test generation. It leverages off DART and other predecessors. Whitebox fuzz testing is much more efficient than the conventional black box methods. It can find several errors that are beyond the reach of the black box fuzzers.

Table 1 : Some recent tools that have used symbolic execution

Tools	Description
Symbolic (Java) PathFinder	It is a test generation system based on the symbolic execution model, based on the NASA Java PathFinder (JPF), which initially used model checking. It implements *generalized symbolic execution*, that adds multi-threading and other capabilities in extension to classical symbolic execution. It has helped discover subtle bugs in several critical NASA systems.
DART	It tries to combine random testing with symbolic execution, to maximize the coverage, while being more efficient as well. It is an example of both symbolic and concrete execution or concolic execution. It was developed for C.
CUTE and jCUTE	It extends DART to handle multi-threaded programs. Can also perform pointer analysis and solves pointer constraints. CUTE is for C whereas jCUTE is for Java.
CREST	It is an open source tool designed for C. It is also extensible and has been used as a basis for several other tools.
SAGE	It is an automated whitebox fuzz tester, that uses compositional symbolic execution. It extends unit testing to whole application testing.
Pex	It is a test generation tool for .NET code. It was developed at Microsoft as a Visual Studio Tool.
EXE	It is a symbolic execution tool for C, written especially to check complex code and low level systems code. It introduces several optimizations, to achieve considerable speed up.
KLEE	It is the successor of KLEE which is based on the LLVM framework. It extends EXE by considering environmental interaction and better memory management for storing states.

Whitebox fuzzing introduces several new concepts to extend symbolic testing to whole applications. It tries to deal with path explosion and other obstacles by introducing a novel parallel state space search algorithm called generational search algorithm. Also because of other technological improvements such as more advanced SMT solvers and concise constraint representations, it can be applied to programs having millions of lines - such as large file parsers, etc.

The SAGE [20] fuzzer based on this technique has become a highly popular tool and is used regularly at Microsoft Corporation. It has saved millions of dollars and found several bugs. It represents the largest ever use of a SMT solver. We examine SAGE in more detail in section 5.

4. Symbolic Execution : Major Challenges and Proposed Solutions

There are several challenges that are an area of concern for symbolic execution. Some of these are enlisted below:

1) How to extend symbolic execution to handle complex and recursive data structures and loops?
2) How to handle multi-threaded and nondeterministic programs with symbolic execution?
3) How to extend the constraint solver to solve string constraints and support other native code features?

A good survey of these and other challenges is presented in [8]. Here we focus on the two major issues that have become a point of focus for researchers. These are:

1) How to curb the path space explosion?
2) How to optimize constraint solving?

Here we present 3 different approaches that attempt to overcome one or both of the above mentioned hurdles.

4.1 RWSet Analysis Based Path Pruning

The idea of RWSet analysis for countering path explosion was proposed in [21].

4.1.1 Approach

Here the authors propose to make symbolic execution more scalable by using the concept of *Read-Write Set (RWSet)* analysis, to discard all paths in the execution tree that will produce the same effect as a previously executed path. Here two major ideas are used to prune the paths. Firstly whenever an execution reaches a program point in exactly the same state as a previous execution then it should be pruned. Secondly if an execution differs from a previous execution only in values that are never going to be written, then it cannot cause a different execution flow, so even it should be discarded. By truncating suffixes of all such paths, significant gains are expected, because at each branch point these suffixes could spawn an exponential number of paths.

4.1.2 Implementation

The *constraint cache* is used to record the states corresponding to which a program state was reached. Now for the currently executing path, if we ever get a cache hit, then obviously the execution of that path should be aborted. The concept of *write set* is used to store only the difference of various concrete states from a common initial state, since storing the complete states may be too expensive. The concept of *read set* is used to maintain the values that will be read beyond the current program point. All other values should be discarded from comparison. Also the cache must be call site sensitive as otherwise a state stored in the cache for one function call may lead us to erroneously discard a path in some completely unrelated function call.

4.1.3 Evaluation

The tool EXE [10] was run, both with RWSet and without it on some benchmarks. It was found that RWSet significantly reduced the number of paths explored to achieve the same coverage. In all cases less then half the number of paths were explored with RWSet, in a few cases the number dropped to as low as just 11% of the original EXE. The overhead of enabling RWSet was also very less; approximately just 4%.

4.2 Parallel Symbolic Execution

Parallelizing the symbolic execution has been proposed in [22].

4.2.1 Approach

The paper proposes to reduce the time spent in exploring paths of the symbolic execution tree, by means of parallelization. The authors use a set of pre conditions to partition the execution tree and distribute the symbolic execution amongst different workers. The partitioning is done such that each worker can independently execute its subtree without any need of interprocess communication (IPC). This is important because otherwise the overhead of IPC may completely negate the gains from parallelization. Initially a shallow form of symbolic execution is run to generate the pre-conditions which will help partition the execution tree. The authors have named this as *Simple Static Partitioning*.

4.2.2 Implementation

The authors attempt to parallelize symbolic execution in a generic manner to make it applicable to different environments such as multi core computers, grid or clouds. The concept of Simple Static Partitioning is used to distribute work amongst different workers. For this first of all an initial set of constraints is generated via shallow symbolic execution. It is important to obtain a set of constraints that are *disjoint, complete and useful*. This ensures that each worker will do some useful work that does not overlap with any of the other workers. The different partitions that are thus obtained are stored in a queue. The user can control both the depth of the initial symbolic execution and the queue size. Both of these are critical factors that determine the success of the algorithm. The size of the *constraint queue* will determine how effectively the load balancing will be whereas the initial depth will determine the quality of the generated partitions.

4.2.3 Evaluation

While the paper cites several detailed metrics for evaluation, the results in a nutshell show that the speed up is possible in all systems. The results also show that the *number of parallel workers (NPW)* and the constraint queue size affect the speed up. There was 90x speedup in analysis time with NPW = 128 and 70x speed up in automatic test generation, with NPW = 64.

4.3 Memoized Symbolic Execution

In the final approach we present, the authors [12] use memoization to counter both the path explosion problem and the constraint solving cost.

4.3.1 Approach

The main idea is that usually symbolic execution is run several times, for instance whenever an error is detected, the program is modified and then re-run. The authors attempt to leverage from the information gained during earlier runs. To this end they maintain a trie structure, that helps in recording efficiently the states of a particular execution. The computations can be re-used during the next run. For example paths that were not useful in prior runs may be pruned and the constraint solver may be turned off for constraints that were previously solved. At each run the trie structure must also be appropriately modified.

4.3.2 Implementation

The trie data structure is used to store the information from each run. A new trie node is created whenever a branch is encountered during the symbolic execution. The node is very lightweight - storing just the method, and instruction offset and the choice taken. The trie is first of all initialized. During the first execution the trie is constructed to guide future executions. The nodes in the trie are split into bound nodes and unsatisfiable nodes. This trie can later be searched using either BFS or DFS. During the memoized analysis phase, the trie is loaded into the memory. It guides the execution; it is appropriately modified and also compressed to remove irrelevant features in the current context. Finally in the trie merging phase the compressed trie may be optionally merged with the older trie to obtain the complete trie.

4.3.3 Evaluation

It was found that the savings obtained depend on how the program is changed and where it is changed, during for e.g. regression analysis. During each execution, however, it was observed that by memoization there was a dramatic drop in the number of calls to the solver. However while this did not reflect significant time gains for simple-to-analyze programs, the authors predict that memoization may be very useful for critical and complex programs.

5. How applicable is Static Analysis in the real world?

In this section, we discuss three static analysis tools to see how useful they actually are in practice.

5.1 The FindBugs Experience

FindBugs is an open source static analysis tool that was developed at the University of Maryland. In the year 2009, Google held a large scale FindBugs Fixit. This involved over 300 engineers, reviewing thousands of warnings. This was an attempt to evaluate how effective FindBugs would be in practice. Over the course of two days, it was found that static analysis does catch mistakes. However most of these are not very important. Static analysis may at best catch about 5% to 10% of software quality problems. However it is cheaper. If used at an earlier stage static analysis can reduce the development cost. Overall however the results of the review were disappointing, as most of the issues reported by the tool were low on priority and did no cause any significant misbehavior.

5.2 Coverity: Static Analysis in Real World

In this popular paper [23] that was downloaded a record number of times from the CACM website, the authors provide some interesting analyses of why static analysis tools fail to do well in practice.

- Firstly to make a tool popular some sort of trial run or demo must be given to customers. Many a times the customer's do not allow the tool to access some of their code for safety and privacy concerns. Thus the tool is unable to find the serious or important bugs and fails to convince the potential customers.
- Many a times the compilers that the company uses to write the code are in-fact buggy. They often deviate from the laid down standards for languages like C or Java. This leads to totally unexpected behavior in some cases when the tool is used along with such compilers. However the companies are unwilling to make any changes in their development environment which forces some ugly workarounds.
- The programmers often don't take the bugs reported by the tool seriously. If they can't understand a reported bug they pass it off as a false positive, rather than trying to analyze the code base.
- Often the bugs reported by two or more static analysis tools are different. Often the static tools themselves are erroneous so that the developers don't have much faith in them.
- If the tool reports false positives of more than 30% then people generally tend to ignore the tool.
- Often the programmers write up any sort of junk for the code. The tool must be designed to take into consideration some of the foolishest errors.

5.3 SAGE : Whitebox Fuzz Tester

In this paper [20] again from the CACM, the picture is remarkably different and rosy from the previous two experiences. We have already described white box fuzzing inan earlier chapter. SAGE the tool designed, based on this

has made a massive impact in the development scenario at Microsoft. We list some of these below:

- SAGE has been running 24/7 on 100+ machines since 2008.
- Since SAGE has extended testing to whole application level, it has found hundreds of bugs in apps, media players, etc that were missed by everything else.
- It has found 33
- Due to SAGE millions of dollars lost in bugs and patches have been saved.
- It holds the record for the largest computational use of the SMT solvers.
- SAGE is today used daily in many groups at Microsoft. It is easy to deploy and fully automated.

This shows that when static analysis is combined with symbolic execution, there may possibly be dramatic rise in applicability.

6. Conclusion & Research Directions

In this paper, we discuss program testing and verification, focusing on symbolic execution. We explore tools - KLEE and Kudzu that apply symbolic execution, next we look at the upcoming paradigms in symbolic execution - concolic and compositional execution. Finally, we have highlighted major challenges to symbolic execution and three possible solutions namely - parallelization, memoization and pruning, to address them. Research directions to go from here are toward improving the scalability of symbolic execution, and heuristic searches that will help to explore more interesting program states. Another interesting area would be to develop more generalized and powerful constraint solvers.

References

[1] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385-394, Jul. 1976.

[2] F. Nielson, *Principles of program analysis*. Berlin New York: Springer, 1999.

[3] S. S. Muchnick, *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[4] P. Cousot, "Abstract interpretation in a nutshell," howpublished, 7th October 2012. [Online]. Available: http://www.di.ens.fr/cousot/AI/IntroAbsInt.html

[5] F. J, "Symbolic execution," University of Maryland at College Park. [Online]. Available: http://www.cs.umd.edu/class/fall2012/cmsc498L/materials/se.pdf

[6] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask," in *In Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[7] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. ACM, 2011, pp. 1066-1071.

[8] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 4, pp. 339-353, Oct. 2009.

[9] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. USENIX Association, 2008, pp. 209–224.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 10:1-10:38, Dec. 2008.

[11] C. C, "Klee : Effective testing of systems programs," PowerPoint, Stanford University, 16th April 2009. [Online]. Available: www.doc.ic.ac.uk/ cristic/talks/kleestanford-2009.ppsx

[12] G. Yang, C. S. Păsăreanu, and S. Khurshid, "Memoized symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. ACM, 2012, pp. 144-154.

[13] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Security and Privacy (SP), 2010 IEEE Symposium on*, may 2010, pp. 513-528.

[14] R. Majumdar, I. Saha, K. C. Shashidhar, and Z. Wang, "Clse: closed-loop symbolic execution," in *Proceedings of the 4th international conference on NASA Formal Methods*, ser. NFM'12. Springer-Verlag, 2012, pp. 356-370.

[15] C. Păsăreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 15-26.

[16] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 571-572.

[17] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. ACM, 2005, pp. 213-223.

[18] P. Godefroid, "Compositional dynamic test generation," in *ACM SIGPLAN Notices*, vol. 42, no. 1. ACM, 2007, pp.47-54.

[19] P. Godefroid, M. Levin, D. Molnar *et al.*, "Automated whitebox fuzz testing." NDSS, 2008.

[20] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage:Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20:20-20:27, Jan. 2012.

[21] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 351-366, 2008.

[22] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *Proceedings of the 19th international symposium on Software testing and analysis,* ser. ISSTA '10. ACM, 2010, pp. 183-194.

[23] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66-75, Feb. 2010.

[24] A. Chaudhuri and J. S. Foster, "Symbolic security analysis of ruby-on-rails web applications," in *Proceedings of the 17th ACM*

conference on Computer and communications security, ser. CCS '10. ACM, 2010, pp. 585-594.

[25] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar based whitebox fuzzing," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 206-215.

[26] B. Johnson, "A study on improving static analysis tools: why are we not using them?" in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. IEEE Press, 2012, pp. 1607-1609.

[27] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 241-252.

About the Authors

Dr. Dhiren Patel is currently a Professor of Computer Engineering at NIT Surat. He carries over more than 20 years of experience in Academia, R&D and Secure ICT Infrastructure Design. Prof. Dhiren has authored a book "Information Security: Theory & Practice" published by Prentice Hall of India (PHI) in 2008. He chaired IFIP Trust Management VI conference in 2012. He has been a Visiting Professor at University of Denver Colorado USA (2014) and at IIT Gandhinagar India (2009-11).

Madhura Parikh did her B.Tech in Computer Engineering from NIT Surat (2009-13) and her Masters in Computer Science from The University of Texas at Austin (2013-14). Her interests are in Computer Systems, Machine Learning and Artificial Intelligence. She is also an open source enthusiast.

Reema Patel, Ph.D in Computer Engineering Department, NIT Surat. She received her B.E., and M.Tech., degrees in Computer Engineering from NIT Surat, in 2006 and 2010 respectively. Her main areas of research interests are Formal Verification, Modeling and Analysis Methods for Stochastic Systems and Information Security.