Scalable Automated Testing Using Symbolic Execution

Stefan Bucur
DSLAB, I&C, EPFL

***Abstract*—Current software testing processes involve significant human intervention, which is both error prone and unscalable. Moreover, recent results in automated testing and verification are still infeasible for large real systems.**

In this paper, we review symbolic execution as a promising scalable testing technique by presenting three state-of-the-art testing tools based on it. Our preliminary research results further show that the path explosion problem can be alleviated by parallelizing symbolic execution in large, shared nothing clusters. We also propose new techniques to further improve the parallelism and also make the state exploration more efficient.

***Index Terms*—software testing, symbolic execution, path explosion, thesis proposal, candidacy exam write-up, EDIC, EPFL**

I. Introduction

SOFTWARE testing techniques have not progressed significantly in the last two decades. The approaches used by testing engineers in the '90s are mostly the same as the ones used today, while at the same time software has grown orders of magnitude in terms of size and complexity. This lead to the current situation where up to 50% of a software project resources and developers are dedicated to testing [1], [2]. And even with this amount of effort, the software bug density per lines of code has remained about the same [3], while the testing process efficiency being affected by human omissions and errors.

Proposal submitted to committee: July 5th, 2010; Candidacy exam date: July 12th, 2010; Candidacy exam committee: Martin Odersky, George Candea, Bernard Moret.

This research plan has been approved:

Date: ____________________

Doctoral candidate: ____________________
(S. Bucur) (signature)

Thesis director: ____________________
(G. Candea) (signature)

Thesis co-director: ____________________
(n/a)

Doct. prog. director: ____________________
(R. Urbanke) (signature)

Recent research results attempt to alleviate this problem by automating the testing process. They range in effectiveness depending on the degree of approximation (abstraction) that they offer. For instance, at one end of this range we find static analysis, which is a complete but unsound verification technique (it gives false positives).

Symbolic execution is at the other end of the range of verification techniques. The idea behind symbolic execution is to substitute program inputs with symbolic values that represent "anything." Program operations will then form expressions that contain these values, and branching points will create conditions that must hold, depending on which branch is taken. Symbolic execution maintains state information for each execution path: when a branch is encountered, the current execution state is forked, and the system will follow both paths. A path end (program termination or bug hit) will generate a test case by solving the current path condition. The solution – a concrete input – can be fed back to the program in order to reproduce the execution path and hit the bug.

Symbolic execution is both sound and complete, as it is able to explore every possible execution path of the program. However, the number of paths of a program is at least exponential in the number of branches in the code, and it is often unbounded because of loops. This is commonly known as the "path explosion" problem. Even if recent research results showed that symbolic execution can be effective for finding real software bugs, and that it can be applied to a broad category of software [4], [5], the path explosion problem still limits the size of the code which can be thoroughly explored.

Model checking is a different verification technique that can offer both soundness and completeness. It relies on exploring every possible state that a model can be in, and checking properties of each state. However, since it relies on the existence of a model [6], or focuses only on certain functionality of a program [7], it cannot be used as a general testing tool. Therefore, we consider symbolic execution as the most general and precise method for testing software programs.

Our goal is to enable symbolic execution to scale to arbitrarily large software, whose size goes beyond millions of lines of code, and this paper presents the steps we have taken thus far, together with the research directions from this point on.

First, we explore the state of the art in the use of symbolic execution as an effective testing technique (§II), and discuss their limitations with respect to our goal(§III-A). We chose

three representative papers: the first one presents the implementation of a symbolic execution engine that can work on real software [4]; the last two papers show prominent results of dealing with the path explosion problem: the second [8] proposes a way to summarize the semantics of functions and thus avoid exploring the same code in different parts of the symbolic execution tree, and the third [9] uses domain specific knowledge in order to synthesize higher level input and be able to reach deeper paths by avoiding complicated constraints and dead-end paths due to invalid inputs.

Second, we show our preliminary steps in scaling symbolic execution to large, shared nothing clusters (§III-B). A first step to achieve this is to parallelize symbolic execution and then run it in a cluster or cloud, where resources can be added elastically, as needed. However, there are a number of important challenges associated to this, which we also discuss in this paper.

Third, we present our research directions (§III-C), regarding the use of a portfolio approach to deal with heuristic algorithms involved during symbolic execution, and also ways to apply formal verification results to further reduce the size of the explored state space. Finally, we conclude (§IV).

II. Survey of the Selected Papers

This section summarizes three papers that illustrate the state of the art in software testing using symbolic execution. Each summary is structured as follows. The first part is an abstract of the paper, which illustrates the main idea and highlights significant results. The following parts briefly present the background of the paper, and the design and implementation of the proposed system. Each summary ends with an evaluation part, which shows the most important results obtained during experiments with the system.

A. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Abstract

KLEE is a symbolic execution engine aimed at automatically generating test cases that achieve high code coverage on a wide range of programs. KLEE was successfully used to test the GNU COREUTILS suite, and generated test cases covering on average more than 90% of the code, beating the developers' own hand-written test suite. KLEE found new bugs in COREUTILS, and it was also used to verify the equivalence of the BUSYBOX and COREUTILS utilities, finding errors and inconsistencies.

KLEE Architecture

While previous research results showed improvements for the most important problems of symbolic execution – path explosion and environment modeling – they were not practical for real, widely used system programs. KLEE addressed this issue by employing a number of optimizations and search heuristics, together with a straightforward modeling of the environment, which translated into performance improvements by at least one order of magnitude. These achievements enable it to handle "out-of-the-box" programs in an autonomous fashion, controlled by a simple set of command line options.

At any point during execution, KLEE maintains a number of process states, each consisting of a register file, stack, heap, PC, and path condition. Unlike conventional memory, storage locations in a symbolic state refer to general expressions (represented as syntax trees), instead of raw (concrete) values. The core of KLEE is an interpreter loop which selects a state to run, and then executes a single instruction in the context of that state.

The execution of LLVM instructions on states is straightforward in most cases. For instance, when executing the LLVM "add" instruction:

```
%dst = add i32 %src0, %src1
```

KLEE obtains the expressions associated with the `%src0` and `%src1` values, and constructs a new expression `Add(%src0, %src1)`, which is assigned to `%dst`. For efficiency, in the case both operands are concrete, the expression will be simply replaced with its computed result.

Conditional instructions, as well as operations whose outcome depend on certain conditions, generate state branching points in KLEE. In the case of conditional branches, KLEE queries a constraint solver (STP [10]) to determine if any, or both outcomes are feasible. If both paths are possible, KLEE clones the state and updates the PC and path condition on each path appropriately. Operations that may cause an error also generate an implicit branch that checks for the error condition. For instance, division instructions generate divison by zero checks, while memory accesses are checked against overflows or invalid pointer dereferences.

In addition, memory accesses are handled in a special manner by KLEE, in order to avoid constraint solving bottlenecks. Instead of representing the memory as a large, flat byte array, KLEE maps every memory object in the checked code to a distinct STP array. When a pointer may refer to multiple distinct objects, KLEE will clone the state for each possible case, and constraint the pointer to be within the bounds of its respective object. This method works well in practice, since most of the symbolic pointers may refer to a single object at a time.

KLEE also implements a copy-on-write mechanism at memory object level for compacting the representation of states. The state heap is implemented as an immutable map, such that portions of the heap can be shared between states, and cloning operations take constant time.

Since the cost of constraint solving dominates the symbolic execution time, a series of optimization passes are applied on constraint expressions before being handed to STP. First, expressions are rewritten and simplified according to basic arithmetic rules. Implied values resulting from simple linear equations are also removed and have the equation variables replaced by their concrete solution. Second, stronger constraints replace older, weaker ones, instead of keeping both in the path condition. Also, constraints irrelevant to the current condition are removed from the constraint set before the query is sent to the solver. Finally, a cache that maps constraint sets to previously computed solutions proves to be very useful in practice, since redundant queries are frequent.

Another key mechanism in KLEE is the search strategy.

KLEE selects the state to run at each instruction by interleaving the following two search heuristics: (a) random path selection, which selects states by walking down the symbolic execution tree and picking at random each branch to take next, until a state (tree leaf) is found, and (b) a coverage-optimized search, which selects states likely to cover new code in the immediate future based on heuristics that compute weights for each state, then randomly select a state according to these weights.

Finally, KLEE comes with a model for file system access. The model is written as C code which is linked with the program code and replaces the original behavior of the relevant system calls. File system operations reduce to in-memory writes and reads, which preserve any symbolic information obtained up to the points the operations were invoked.

Evaluation Highlights

KLEE uses code coverage to measure the quality of the generated test cases. Only test cases covering new code are generated during symbolic execution. After that, coverage is measured by using `gcov` on the instrumented original binary, instead of the LLVM bytecode. The evaluation considers only the code in the program itself, ignoring library code.

KLEE was successfully used to test the COREUTILS suite, BUSYBOX, and the HISTAR kernel, where it both achieved more coverage and discovered new bugs.

When run on the COREUTILS suite (89 tools), KLEE obtained on average 90.9% code coverage, with a minimum coverage of 62.6%, and with 16 tools getting 100% coverage. Moreover, the generated test cases beat the manually written test suite that comes with COREUTILS. For instance, overall line coverage in the test suite is only 67.7%, and a single tool (`true`) achieves 100% code coverage. When also compared with a random test suite, KLEE performed significantly better.

KLEE also found ten unique bugs in COREUTILS, most of them being memory error crashes. Three of the errors existed since at least 1992, without being noticed or covered by a test case until now.

Tool equivalence is another powerful use case of KLEE. Two procedures `f` and `f'` that are supposed to implement the same specification can be checked for equivalence by feeding them the same symbolic argument, and then asserting that they return the same value: `assert(f(x) == f'(x))`. KLEE would then check each path that reaches this assertion and try to produce a test case demonstrating the differences, if they exist. This approach was successfully used to find many differences between COREUTILS and their equivalents in BUSYBOX.

B. Demand-Driven Compositional Symbolic Execution

Abstract

This paper shows a way of performing symbolic execution of large programs in a compositional and demand-driven manner. Compositional symbolic execution finds complete, *inter*procedural paths by combining feasible *intra*procedural paths. “Demand-driven” means that only the intraprocedural paths that help leading to a certain goal (e.g. an assertion) are executed. The symbolic execution is performed using first-order logic, with an off-the-shelf SMT solver [11]. The prototype is implemented as an extension of Pex [12], a testing framework for .NET programs. Preliminary results show that the new technique was able to reach assertion violations which were far beyond the scope of non-compositional symbolic execution.

Overview

The idea behind compositional symbolic execution is to maintain a set of symbolic trees, representing summaries for each function in the program (see Fig. 1). At each point during symbolic execution, a function tree (summary) indicates which paths were explored inside the function. The unexplored paths are marked with ”dangling” nodes (the circles in the figure). A complete, interprocedural path would be obtained by concatenating intraprocedural paths which do not end in a dangling node, except for the last path component. If dangling nodes are allowed inside a path, that path will be interrupted, and solving the associated constraints will not guarantee its replay.

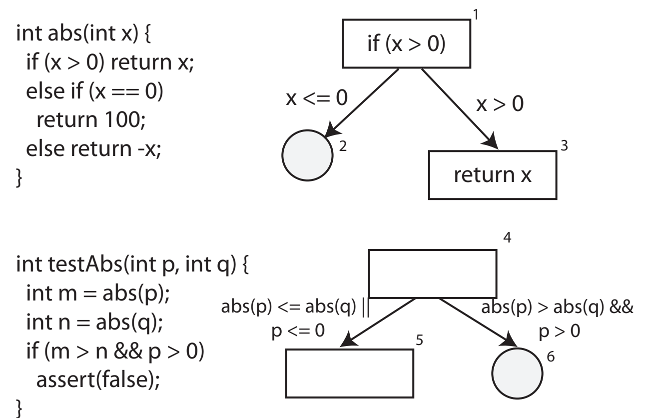

Fig. 1. Example program, together with execution trees for the functions `abs` and `testAbs` exercised with inputs 1 and (1, 1), respectively.

The compositional approach permits two key properties of the execution: lazy exploration, and relevant exploration. Lazy exploration aims to explore as few paths as possible – when a new node needs to be explored, the algorithm will favor currently known paths instead of searching new paths and increase the size of the summaries. In case there is no existing path to the goal, relevant exploration helps by driving the exploration to paths that end in the goal, and thus automatically prune entire sub-search trees.

The main algorithm starts with an empty set of intraprocedural execution trees, and a random program input. Symbolic execution is performed iteratively, in a concolic [13] manner. At each step of the iteration, the current concrete input is used to execute the program along a certain path. At the same time, the code is executed symbolically, the path constraints are collected and the intraprocedural trees are augmented accordingly. For the example in the figure, the illustrated trees correspond with running the programs with the input pair $(1, 1)$.

At the end of each iteration, a new dangling node is chosen according to a search heuristic, its interprocedural constraints are computed, and then sent to the SMT solver. If the solver finds the path unfeasible, the node is removed, otherwise a new iteration is started with the new input computed by the solver. The next subsections present the approach in more detail.

Compositional Symbolic Execution

In compositional symbolic execution, the global constraints on a node are a conjunction between a calling context – the condition to enter the function that contains the node – together with a local (intraprocedural) path constraint. This subsection briefly explains each predicate involved in the final global path constraint formula. For a complete, formal specification of each predicate, please refer to the original paper [8].

The *local path constraint* of a node n in a function f is defined as the path constraint of the path w in the local tree $\mathcal{T}_f$. This constraint is a conjunction of all the constraints appearing on the edges of the path w (resulted from branching in the code), together with a conjunction of the (possibly partial) summaries of the functions invoked along w. The summaries are expressed using the so called definition predicates.

The *definition predicate* describes the semantics of the function as it is currently represented by the partial summary. It is a disjunction of conditions across all the terminal nodes currently present in the summary. For function return points, the condition represents the local path constraint, together with a return value assignment. For a dangling node, the condition is a conjunction between the local path constraint and a special auxiliary boolean variable associated with that node. If the variable is forcefully assigned to a value, it can control the feasibility of the local path. The role of the dangling node variables in the symbolic execution algorithm is explained in §II-B.

The *calling context predicate* associated with a function f describes under which conditions and with which arguments f can be reached. If f is the entry function, then the predicate specifies that its arguments are the actual program inputs (symbolic values). Otherwise, the predicate includes a disjunction of conditions for all the callsites of f. If f is called on a path w in a function g then the condition associated with this callsite is a conjunction between the calling context of g itself, together with the local path constraint of w. The calling context predicate of a function also includes calling sites from dangling nodes – in this case, the feasibility of such sites is controlled by the auxiliary boolean variables included in the predicate.

By using the three types of predicates presented before, we can define the global interprocedural path constraint for a node n in the execution tree $\mathcal{T}_f$ of a function f. It is a recursive formula which represents the disjunction of path constraints of all the interprocedural paths to n that can be formed by joining intraprocedural paths, in the execution trees of different functions.

The Demand-Driven Approach

In order to allow symbolic execution to be demand-driven, paths are permitted to be interrupted by combining intraprocedural paths that end in dangling nodes. These paths are called partially explored, as opposed to fully explored ones, which may end at a dangling node, but not go through one.

The demand-driven algorithm implements lazy exploration by first attempting to build a continuous path using the already explored nodes in the intraprocedural trees. Fully explored paths have constraints which generate inputs that guarantee that the new execution will reach the goal (here we also assume that the SMT solver is sound and complete – it has perfect precision). In this case, the auxiliary boolean variables associated with dangling nodes are forced to be false, except for the ending node. This ensures that the constraint solver will not find inputs that go through dangling nodes.

Relevant exploration is achieved by constructing incomplete paths that end in the point of the exploration goal (an intraprocedural node). If lazy exploration is not possible, the auxiliary boolean variables are not constrained any more, and the solver returns a program input that may exercise a path to the target. However, such a path is not guaranteed to reach the target, since the program behavior beyond the dangling nodes is unknown.

Preliminary Experiments

A prototype of this approach was implemented on top of the Pex [12] testing framework for .NET programs, using the Z3 [11] SMT solver.

Experiments were run on three programs: HWM – a string manipulation program, a Pascal-like language parser, and a simple increment-decrement program. Each program had an assertion at the end, that needed to be reached and proven to be failed. Demand driven compositional symbolic execution was compared with the classic, non-compositional execution, on the three evaluation cases.

Results showed that classic symbolic execution needed orders of magnitude more iterations to reach the assertion, and in the case of HWM, it even timed out after 20000 executions, without any success. Finally, when the symbolic execution was demand-driven and compositional, the time per execution was significantly longer, as the formulas generated are more complicated and the constraint solver needs more time. However, the total time required to reach the goal was still one order of magnitude lower, because of the improvements in exploration precision offered by lazy and relevant explorations.

C. Grammar-Based Whitebox Fuzzing

Abstract

Whitebox fuzzing is a form of test case generation based on mixing concrete and symbolic execution, designed for security testing of large applications. This technique poses severe limitations when applied on programs with highly structured inputs, such as compilers and interpreters, which process their inputs in stages.

Grammar-based whitebox fuzzing helps exercising paths at all stages by adapting symbolic execution to directly generate grammar-based constraints and use a grammar-based constraint solver that generates inputs which avoid dead-ends.

This approach was successfully tested on a large security-critical application – the JavaScript interpreter of Internet Explorer 7. Compared to regular whitebox fuzzing, grammar-based whitebox fuzzing increased the coverage of the code generation module (the deepest in the module hierarchy) from 53% to 81%, while using three times fewer test cases.

From Blackbox to Grammar-Based Whitebox Fuzzing

This subsection explains the concept and the main algorithm behind grammar-based whitebox fuzzing. It starts by presenting the simplest forms of fuzzing, and then gradually shifting to the more advanced techniques of whitebox fuzzing.

Blackbox fuzzing is the simplest form of fuzzing, consisting of randomly modifying well-formed inputs, and using them as test cases. An improved version of this technique uses a grammar to generate the original well-formed inputs, and it can also act as a heuristic for guiding the generation of the random variations.

A more recent alternative, *whitebox fuzzing*, executes the program both concretely and symbolically. It starts with an initial, well-formed input, and executes the program concretely, but at the same time it collects all the constraints generated by the encountered branches along the execution path. Each constraint is then negated one at a time, and a constraint solver is then used to generate a new input, which is added back to a queue of inputs. Then one input is extracted from the queue, and the process is repeated.

The goal is to exercise as many different paths as possible, and find bugs as fast as possible using various search heuristics. The algorithm terminates when a testing time budget expires, or no more inputs can be generated.

The effectiveness of whitebox fuzzing is limited when testing applications with highly structured inputs, such as compilers and interpreters. In that case, the programs have multiple processing stages, and due to the enormous number of control paths in early stages, whitebox fuzzing rarely reaches later stages of the program. Moreover, in addition to path explosion, symbolic execution itself may be bottlenecked in the first stages because of constraint complexity.

Grammar-based whitebox fuzzing enhances whitebox fuzzing with a grammar-based specification of inputs. It has two components: (a) a generator of higher-level symbolic constraints expressed in terms of symbolic grammar tokens, instead of original input bytes, and (b) a custom constraint solver capable of solving constraints on symbolic tokens and offering solutions which are also accepted by a given, context-free grammar.

In the original case of whitebox fuzzing, the symbolic values in the path constraints were the input bytes of the program. In the grammar-based extension, the execution engine marks as symbolic the tokens returned from a tokenization function inside the program. Symbolic execution will then track the influence of these tokens on the control path taken by the program.

This new approach also requires that the constraint solver be aware of the grammar $\mathcal{G}$ of the language, and generate a valid concrete input when one of the constraints is negated.

Context-Free Constraint Solver

Formally, the context-free constraint solver invoked by grammar-based whitebox fuzzing checks whether the language $L(pc)$ of inputs satisfying the path constraint pc contains an input conforming to the grammar $\mathcal{G}$. By construction, the language $L(pc)$ is regular, since each token belongs from a finite set $\mathcal{T}$ of tokens. If $\mathcal{G}$ is context-free, then the language intersection with $L(pc)$ is decidable and there is a sound and complete algorithm for computing the intersection.

The algorithm chosen for the constraint solver is not polynomial in general, it gives satisfactory results in practice for small values of n (less than 60).

The procedure starts by first grouping all the constraints according to the token they refer to. Tokens that appear in the constraints are sorted in the order they were emitted by the lexer. Let t_i be the i-th token emitted, and n the total number of tokens.

The algorithm then iterates n times (for each token in the sequence), and filters the productions from the original grammar that do not match the path constraints for the current token. At each point during the iteration, the algorithm maintains a worklist of productions, initialized with the starting production. When a new token t_i is processed, productions may need to be unrolled in order to uncover all the possible i-th terminals of the context-free language. The unrollings that do not match the constraints for t_i are pruned, and the iteration continues with the next token. Note that since the tokens are traversed in the order of emission, no non-terminals may be found before the i-th position in the unrolling, but only after it.

After the unrolling and pruning, the algorithm checks the emptiness of the resulting language $\mathcal{G}'$, and if the grammar is not empty (the constraint is satisfiable with respect to $\mathcal{G}$), it generates a string from $\mathcal{G}'$.

One observation is that grammars used by the solver may approximate the set of all parsable inputs. This happens if the parsers actually implement additional validation, such as simple type checking, or context sensitive verification, which would be more difficult to capture in the grammar specification passed to the constraint solver. In this case, some generated inputs may be rejected by the actual parser.

Grammar-based whitebox fuzzing also requires a certain amount of domain knowledge: the formal grammar, the location of the tokenization function, and also a reverse tokenization function to generate the raw input string from the synthesized token stream.

Finally, using a grammar to filter out invalid inputs may reduce the coverage in the lexer and parser components, since the generated inputs will not reach paths that handle invalid inputs in those stages. However, experiments show that grammar-based whitebox fuzzing does not decrease the coverage – this is plausible since the actual lexer and parser are used during concrete program executions.

Evaluation

Grammar-based whitebox fuzzing was tested with the JavaScript interpreter in Internet Explorer 7, together with other 4 fuzzing strategies: blackbox, grammar-based blackbox,

plain whitebox, and whitebox+tokens (only the lexical part of the grammar). For all the strategies, 50 seed inputs with 15 to 20 tokens generated randomly from the grammar were used. The size of inputs were chosen as a compromise between input complexity and the rejection rate by the code generator component. Since the inputs generated for the parser do not necessarily make sense at a semantic level, the longer the input, the less likely it was to be accepted.

Each strategy was allowed to run for 2 hours, time which included all the experimental tasks (program execution, symbolic execution, constraint solving, etc.). In order to see whether giving more time would change the results, each strategy was then allowed to run until instruction coverage did not increase during the last 10 hours.

After two hours of execution, grammar-based whitebox fuzzing achieved the best total coverage, as well as the best coverage in the code generator, the deepest examined module. It also achieved the results closest to the manual test suite, which provides the best coverage (but at the same time, it took many man-months to develop). When left running for a longer time, grammar-based whitebox managed to keep covering new code for 97 hours, while the other strategies stalled at 84 hours or earlier.

In addition, grammar-based whitebox fuzzing creates test inputs of the highest quality among the analyzed strategies. This is backed by the fact that the inputs generated by grammar-based whitebox cover most of the instructions covered by the inputs generated with other strategies, while also covering many other instructions.

III. Research Proposal

A. Current State of The Art and Limitations

The three papers presented in the previous section showed different scenarios where symbolic execution was leveraged and how the path explosion problem was alleviated; each approach also has its limitations. KLEE was the closest to apply symbolic execution in its original form – fully symbolic and working with the entire symbolic execution tree. It also employed optimizations and heuristics at the constraint solving level. In a sense, it bridged the gap between the theoretical results of formal verification research, and what it takes to build an engine that runs on real programs, written in a widely used programming language. However, KLEE does not yet scale to programs larger than a couple of thousands of lines of code.

The other two systems addressed path explosion in different ways. Demand-driven compositional symbolic execution fragments the tree into a set of summaries for each program function, and composes complete path constraints by concatenating constraints of intraprocedural paths. This does not eliminate the path explosion problem, since now formulas are much more complex and the constraint solver has a more difficult job to do. However, moving the solving complexity into the constraint solver is a good idea by itself, since modern solvers benefit from years of previous research on optimization techniques, and their efficiency was validated in practice by a wide adoption in the industry. But in the particular case of demand-driven compositional symbolic execution, the formulas contain quantifiers and require a more sophisticated solver than a simple bit-vector and array solver (like KLEE's STP), which can quickly bottleneck the engine.

Finally, grammar-based whitebox fuzzing addresses path explosion by relying on (a) a domain specific knowledge about the program (the context-free grammar specification), together with (b) executing program concretely in order to reach for deeper paths. The first approach limits the applicability of the tool to compiler-like programs, which process their data in stages. Moreover, one needs to come up with a context-free grammar that describes the structure of the data transferred between two stages, which makes the procedure hard or impossible to automate. Second, executing programs concretely for each test case generated is not appropriate when the number of instructions per path is very large. For instance, executing a program that does intensive initialization in the beginning (such as an interpreter that loads libraries) would be more efficient if the initialization would be done only once.

B. Current Work and Preliminary Results

Each of the presented approaches has limited applicability, but their results can help steering our research in the right direction. As our goal is to enable completely automatic software testing for programs of arbitrary sizes, running in any environment, symbolic execution proved to be the most general technique, as it is autonomous, sound and complete. This is why it is our technique of choice for building such a tool.

We believe KLEE to be the closest result that can be used as a starting point for a scalable symbolic execution engine. It was proven to be useful for other testing scenarios [14], [15], and it is also openly available. However, yet a lot of work needs to be done in order to achieve our goal.

Our work thus far focused on building Cloud9 [16], a parallel symbolic execution engine, based on KLEE, that runs on shared-nothing clusters or in a cloud environment. We started from the idea that currently symbolic execution engines are sequential, running on a single machine, so the path explosion bottleneck is hit for relatively low program sizes (thousands of lines of code). Adding more hardware resources (CPU and memory), in the form of individual machines exploring together the symbolic execution tree, would enable more paths – and conversely, larger programs – to be explored in the same amount of time.

However, building a symbolic execution engine to run in a distributed, shared-nothing environment is a challenging task. First, the shape of the symbolic execution tree is not known a priori, so a form of load balancing is required in order to ensure that the nodes do not end up idling, and also in order to avoid performing redundant work (two nodes explore the same portion of the tree). Cloud9 uses a load balancing component, which keeps track of global progress, allocates tasks to workers, and instructs workers to exchange work when load balancing is required.

Second, worker synchronization requires a procedure for encoding and transferring work between nodes. Currently, we employ a state replay technique, which sends fragments of paths to be replayed at destination, instead of serializing

individual states. This has the benefit of avoiding the encoding of environment interactions (e.g. previously run external calls, concrete open files, etc.), and it can also save bandwidth, since multiple paths can be encoded easier and more efficiently than serializing a set of states.

Third, search strategies need to coordinate as well in the distributed environment. In order to achieve this, Cloud9 employs a global progress tracking mechanism that we currently apply for code coverage. Each worker reports to the load balancer its own map of covered lines, and the load balancer aggregates this information and sends back to each worker the current global view. This way, coverage-oriented strategies will avoid searching for already explored code, whether it was discovered locally or by a different worker.

Our prototype implementation operates on infrastructures like Amazon EC2 and Eucalyptus [17]. Preliminary experiments with Cloud9 testing the COREUTILS (up to 4KLOC, each) and the Python interpreter (100KLOC), showed that Cloud9 achieves path coverage scalability. The size of the cloud environment ranged from 1 worker, up to 64 workers.

C. Future Research Directions

Preliminary results showed that our approach scales well in terms of path coverage. However, more work needs to be done in order to enable Cloud9 to scale in terms of code coverage (and other finite coverage metrics), and also improve the effectiveness of finding bugs.

We have identified a number of research directions that are promising, which are described in the following subsections: better exploit the parallelism based on CPU profile, come up with effective search strategies in a parallel environment by adopting the portfolio approach, and finally improve the symbolic execution engine itself through techniques like state merging or abstract interpretation. The order in which they are introduced here is also likely to be the one in which we will address them.

Parallelization Based on Profile Results

Profiling the KLEE symbolic execution engine shows that more than 80% of the CPU time of the engine represents constraint solving. Fig. 2 shows the results of profiling KLEE while testing the `ls` utility. About half of the constraint solving part represents work performed by the STP solver, while the other half is represented by the SAT solver.

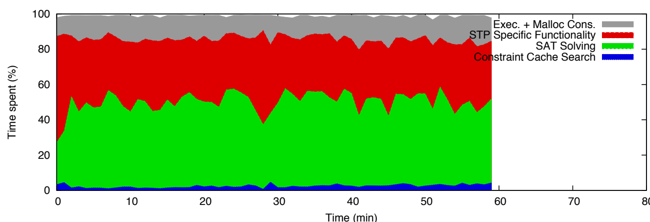

Fig. 2. Results of profiling KLEE's execution, while testing the `ls` utility.

This observation suggests that the constraint solving part of symbolic execution may be more effective and easier to scale than the entire symbolic execution engine. This is also partly because a centralized symbolic execution engine may better benefit from the constraint caching component, which in Cloud9 is currently spanned across all workers.

Parallelizing constraint solving can be done in a number of ways. First, multiple nodes can handle different constraints received from the symbolic execution engine. In this case, a batching mechanism that collects constraints from multiple states at a time and solves queries simultaneously would be more effective than a pipeline handling one state at a time.

Another approach is to have a team of solvers independently solving the same constraint. The first solver to finish would provide the solution back to the symbolic execution engine – this technique is called a *portfolio approach*, and it is described in more detail in the next subsection, which also discusses different usage scenarios for portfolios in Cloud9.

The Portfolio Approach

The portfolio idea originates from the financial field, as the Modern Portfolio Theory (MPT) [18]. The basic principles in MPT are diversification and periodic rebalancing.

In the case of symbolic execution, we identified two major uses of portfolios. The first one is for the search strategies used in Cloud9. Instead of having a single strategy in the system, each worker would employ a number of strategies, and send updates per strategy to the load balancer. The load balancer would aggregate the statistics and observe which strategies performed better in terms of the search goal (e.g. coverage or bugs), and then decide in which one to re-invest more cluster resources.

The second use case is the constraint solver component. Previous research [19] showed that constraint solvers can give different performance results depending on the seed values used for random decisions. This was exploited in a system that ran the same solver on multiple nodes, but with different seed values. During the same testing period, the portfolio managed to solve significantly more constraints than a single, sequential instance of the same solver.

In addition to those results, our own experiments showed that a heterogeneous set of solvers offered good portfolio balance in solving constraints generated by KLEE during the symbolic execution of some COREUTILS. Fig. 3 shows the portfolio winners (the solvers with the lower solving time) for the SAT formulas generated by KLEE while testing the `ls` utility. Three state-of-the-art solvers, MiniSAT, PrecoSAT and Clasp, were aggregated in a portfolio and KLEE was modified to externally redirect SAT formula requests.

Symbolic Execution Improvements

Finally, another way to significantly reduce path explosion is to modify the semantics of the symbolic execution engine and eliminate states by either approximating constraints (in the sense given by abstract interpretation [20]), or by merging compatible states in a single, larger one.

In the first case, abstract interpretation is known for the fact that it can converge to a solution in a finite amount of time (depending mostly on the size of the program), if the state abstractions are properly chosen. Currently, symbolic execution in KLEE is both sound and precise, so it does not

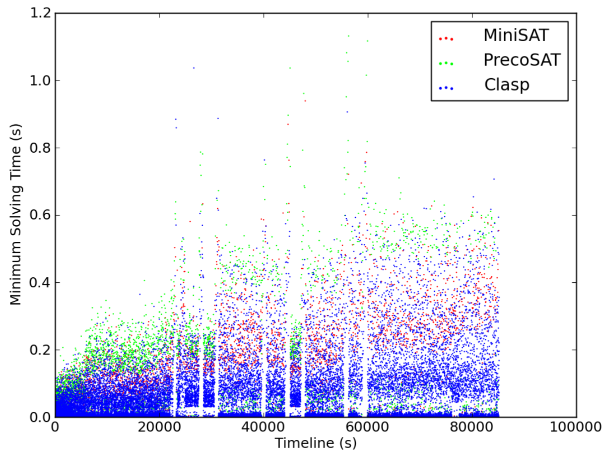

Fig. 3. The SAT portfolio winners while testing the `ls` utility with KLEE having the portfolio as an external solver.

allow approximations. However, one can selectively relax the constraints in points where path explosion would not permit an acceptable progress rate. An abstract interpretation engine would then realize that states have converged to a fix-point in the piece of code of interest, and then decide to prune any other path that would leave the states invariant.

In the second case, state merging preserves both soundness and completeness. State merging can be done when two different states are at the same point in the control flow graph, and their contents are compatible (e.g. same symbolic memory structure). In this case, the corresponding memory locations that are identical will be shared by the new state as well, and the ones that are different are replaced by an if-then-else expression conditioned by a new free symbolic value, and whose branches are the two different state values.

While this simplifies the work of the symbolic execution engine and may prune entire search sub-trees, it is merely a shift of the problem complexity from the engine to the constraint solver. An "or" expression added in a constraint has the potential of doubling the size of the elementary SAT formula sent to the SAT solver, hence canceling any benefit from eliminating a path. However, advancements in SAT solving and the inherent simpler representation of constraints at that level make it likely that the heuristics in the SAT solver will find a solution faster than a symbolic execution engine.

IV. Conclusions

This paper presented symbolic execution as a potential technique for enabling scalable automatic testing of software.

In the first part we presented three state-of-the-art techniques that leverage symbolic execution, and discussed their design, implementation, and how they perform in practice. First, KLEE is a symbolic execution engine aimed at automatically generating test cases that achieve high code coverage on a wide range of programs. Second, compositional demand-driven symbolic execution keeps execution trees for each function in the program, and finds complete execution paths by combining feasible intraprocedural paths. Thus it is able to prune paths corresponding to already explored code, and effectively obtain symbolic summaries for functions. Third, grammar-based whitebox fuzzing helps exercising paths at all stages of a program (e.g. compiler) by adapting symbolic execution to directly generate grammar-based constraints and use a grammar-based constraint solver that generates inputs which avoid dead-ends.

Each of these approaches have limitations that prevent them from being practical for any type of programs. Klee and demand driven compositional symbolic execution do not scale for programs of more than a couple of thousands of lines of code, while grammar-based whitebox fuzzing requires domain specific knowledge about the program.

In the second part we showed our preliminary results with Cloud9, a parallel symbolic execution engine, that runs in a shared nothing cluster. We then proposed three research directions towards scaling even more symbolic execution: better exploit the parallelism based on CPU profile, come up with effective search strategies in a parallel environment by adopting the portfolio approach, and finally improve the symbolic execution engine itself through techniques like state merging or abstract interpretation.

References

[1] G. Fohler, *Embedded Systems Design - The ARTIST Roadmap for Research and Development*, ser. Lecture Notes in Computer Science. Springer Verlag, January 2005, ch. Adaptive Real-time Systems (contribution).

[2] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Tech. Rep., 2002.

[3] S. McConnell, *Code Complete*. Microsoft Press, 2004.

[4] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[5] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with DDT," in *USENIX*, 2010.

[6] G. J. Holzmann and D. Bosnacki, "Multi-core model checking with SPIN," in *IPDPS*, 2007.

[7] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "MoDist: Transparent model checking of unmodified distributed systems," in *NSDI*, 2009.

[8] S. An, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *In Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008.

[9] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI*, 2008.

[10] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV*, 2007.

[11] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008.

[12] N. Tillmann and W. Schulte, "Pex," http://research.microsoft.com/Pex.

[13] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *FSE*, 2005.

[14] V. Chipounov and G. Candea, "Reverse engineering of binary device drivers with RevNIC," in *EuroSys*, 2010.

[15] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated debugging," in *EuroSys*, 2010.

[16] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," in *LADIS*, 2009.

[17] "Eucalyptus software," http://open.eucalyptus.com/.

[18] H. Markowitz, "Portfolio selection," *The Journal of Finance*, vol. 7, no. 1, pp. 77–91, 1952. [Online]. Available: http://www.jstor.org/stable/2975974

[19] L. Bordeaux, Y. Hamadi, and H. Samulowitz, "Experiments with massively parallel constraint solving," in *IJCAI*, 2009.

[20] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.