Advances in Symbolic Execution

Guowei Yang*, Antonio Filieri†, Mateus Borges†, Donato Clun†, Junye Wen*
*Texas State University, USA, {gyang, j_w236}@txstate.edu
†Imperial College London, UK, {a.filieri, m.borges, d.clun16}@imperial.ac.uk

***Abstract*—Symbolic execution is systematic technique for checking programs, which forms a basis for various software testing and verification techniques. It provides a powerful analysis in principle but remains challenging to scale and generalize symbolic execution in practice. This chapter reviews the cutting-edge research accomplishments in addressing these challenges in the last five years, including advancements in addressing the scalability challenges such as constraint solving and path explosion, as well as advancements in application of symbolic execution in testing, security, and probabilistic program analysis.**

***Keywords*-constraint solving; path explosion; incremental checking; parallel analysis; compositional analysis; memory modeling; concurrency; test case generation; fuzzing; probabilistic analysis.**

I. Introduction

Symbolic execution is a program analysis technique enabling the exploration and summarization of a large number of execution paths by replacing program inputs with symbolic parameters and studying the conditions on these parameters that determine the execution of each element of the program [96].

A symbolic execution engine implements two complementary features. First, it maintains a symbolic representation of the program state, which is updated after the execution of each instruction according to the semantics of the programming language used to implement the program; depending on the language, the program state may be include complex information, like a symbolic representation of the heap or the state of multiple threads and their scheduler. Second, it uses a constraint solver to determine if the evaluation of a conditional branch in the current symbolic state can be satisfied; this is critical to decide if there exists a concrete assignment to the symbolic variables of a program that can actually exercise the current execution path. Both these features place critical challenges to the applicability of symbolic execution in practice.

In this chapter, we review the main results in symbolic execution achieved in the last five years. By focusing on the current edge of research, we aim at providing an overview of the latest challenges elicited by the research community to improve the scalability and generality of symbolic execution.

After introducing the necessary definitions and background in Section II, we report on recent results in constraint solving that directly enabled significative advancement in symbolic execution. These include both reasoning techniques for domains and theories specifically relevant for symbolic execution, e.g., strings or non-linear numerical constraints, and techniques aiming at improving the practical scalability of constraint solver, from caching and reuse to heuristics enabling faster solutions for specific constraint patterns arising in symbolic execution.

We discuss the problem of *path explosion* and the most recent mitigation strategies in Section IV. These strategies include, among the others, the selective concretization of certain execution paths, the use of loop summarization to prevent unnecessary loop unwinding of loop, and the use of directed and incremental search strategies to maximize the exploration effectiveness of partial symbolic execution given the available computational resources and time.

Section V reviews recent advancement in compositional symbolic execution techniques, which allow a principled trade-off between the accuracy of local analyses and the reuse of partial results, which may also enable the parallelization of several analyses.

Symbolic data structures and heap representations are reviewed in Section VI, where we overview both the formalization of heap objects' structure and properties for automatic reasoning and the challenges of capturing the semantics of memory accesses via symbolic object references.

Section VII concludes the overview of the main across-the-board challenges of symbolic execution by discussing the recent advancements on modeling concurrency, both for detecting concurrency bugs and for improving the scalability of symbolic execution engines in presence of the combinatorial explosion of execution paths due to the interleaving of multiple threads.

Sections VIII, IX, and X presents the recent advancements in three application areas of symbolic execution that have seen significant advancements in the last five years. In particular, Section VIII discusses how automated test case generation and test suite optimization benefited from advancements in constraint solving, hybrid concrete/symbolic execution, directed symbolic execution, and the symbolic modeling of non-standard programming artifacts like databases. Section IX, presents a set of symbolic/hybrid execution techniques for the detection, automatic exploitation, and automated repair of security vulnerabilities. Finally, Section X, summarizes a new research thread on probabilistic symbolic execution, which extends classic symbolic execution to quantify the probability of executing each program element instead of simply assessing the possibility of such element being executed. This paves the way to new applications of symbolic execution for the analysis of quantitative program properties, including performance, information leakage, or reliability.

Finally, in Section XI we include a list of currently maintained symbolic execution tools, sketching for tool the languages it supports and the problems it is optimized for. We conclude with some final remarks in Section XII.

II. Background

```
1 int compute(int curr, int thresh, int step){
2   int delta = 0;
3   if (curr < thresh){
4     delta = thresh - curr;
5     if ((curr + step) < thresh)
6       return -delta;
7     else
8       return 0;
9   } else {
10      int counter = 0;
11      while (curr >= thresh) {
12        curr = curr - step;
13        counter++;
14      }
15      return counter;
16  }
17 }
```

Fig. 1. Example program

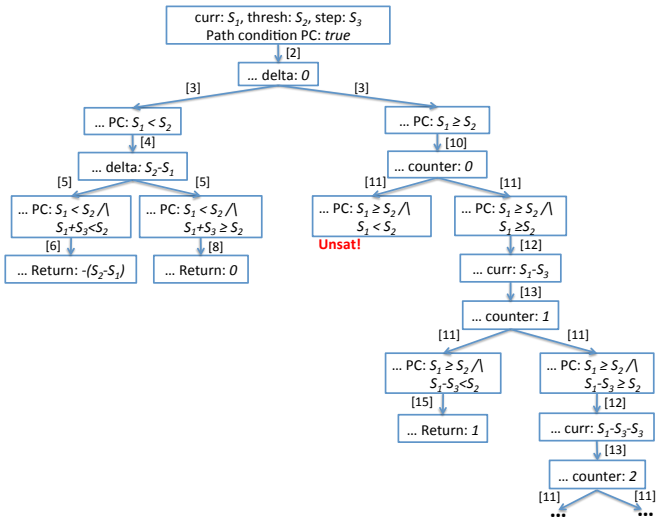

Fig. 2. Symbolic execution tree

Symbolic execution [47], [96] is a powerful program analysis technique for systematic exploration of a large number of program execution paths. It provides a basis for various software testing and verification techniques. The key idea is to use symbolic values in place of concrete values as inputs to execute the program, and to compute resulting output as a function of the symbolic inputs.

A symbolic program state includes the (symbolic) values of program variables and a path condition (PC). The path condition is a (quantifier free) Boolean formula over the symbolic inputs, collecting constraints on the inputs in order for an execution to follow the associated path. Path conditions are checked for satisfiability using off-the-shelf decision procedures [97] during symbolic execution; each time the path condition is updated, it is checked to determine the feasibility of the path; if a path condition becomes unsatisfiable, which means the corresponding path is infeasible, symbolic execution stops exploration of that path and backtracks. A *symbolic execution tree* characterizes all the paths explored during the symbolic execution. Each node represents a symbolic program state and each arc represents a transitions between two states.

We illustrate symbolic execution with the example program shown Figure 1, where the method `compute` has three integer inputs: `curr` (current), `thresh` (threshold) and `step`; it calculates the relationship between the current and the threshold, in increments given by the step value. Its corresponding symbolic execution tree is shown in Figure 2. The path condition PC is initialized as *true*, and the three input variables `curr`, `thresh` and `step` have symbolic values S_1, S_2 and S_3, respectively. Program variables are then represented by expressions in terms of these symbolic inputs; e.g., after executing statement 4, `delta` becomes $S_2 - S_1$. At each branch point, there is a *choice* in the execution and PC is updated with assumptions about the inputs, to choose between alternative paths. For example, after statement 3 is executed, both `then` and `else` alternatives of the `if` statement are possible, and PC is updated accordingly and checked for satisfiability. If PC becomes false, which means there are no inputs to satisfy it and thus the state is un-reachable, symbolic execution does not continue for that path. For instance, when the `while` statement at line 11 is executed the first time, the PC corresponding to exiting the loop is unsatisfiable, and symbolic execution does not continue on that path. The PCs for the explored program paths during symbolic execution can be solved by a constraint solver and the solutions can be formed as test inputs——the execution of the program on these concrete inputs will follow the same path as the symbolic execution.

For programs with loops or recursion, symbolic execution may result in an infinite symbolic execution tree. For example, in Figure 2, the expansion of the right-most leaf in the tree may continue forever. To address this problem, a limit (e.g., a depth bound or a time bound) is typically put on the search for symbolic execution.

Classic symbolic execution statically analyzes programs without any concrete executions, and in practice it can be challenging to apply it, for example, for programs with invocations to untraceable libraries/native code or with constraints that are hard to solve. Dynamic symbolic execution [34], [68], [154], [168], which is also called concolic execution, combines concrete executions and symbolic analysis to mitigate this problem. It starts with a concrete execution on some given or random inputs and collects the PC along the executed path at the same time. The constraints in the PC can be negated so that the resulting PCs are solved by a constraint solver to find inputs to explore alternative paths. This process is repeated until the desired testing coverage has been achieved.

III. Constraint Solving

Symbolic execution relies on *constraint solvers* to decide the satisfiability of path conditions and thus avoid exploring unfeasible paths. Although satisfiability is a hard (NP-complete) [19] problem, algorithmic and engineering advances in constraint solving over the last few decades made it more manageable. One of the main developments is the creation of efficient *Satisfiability Modulo Theories* (SMT) solvers. SMT

solvers combine together specialized decision procedures for specific *theories* (i.e. a well-defined interpretation of a set of symbols in a first-order logic formula) with a core SAT solver. This combination allows to express and decide constraints that depend on multiple theories. Developers of symbolic execution engines must choose carefully the theory, and consequently the solver, used to model and decide the constraints encountered during the exploration of the program. For example, the code in Figure 1 only performs addition and subtraction over the integers, which can be handled by the theory of linear integer arithmetic. However, this representation may miss edge-case behaviors like arithmetic overflows since `int`s are represented in hardware as 32-bit values. Modeling the constraints in the theory of *bitvectors* will ensure that the formula semantics will match the actual execution in the hardware.

A. Simplification, reuse and caching

Symbolic execution of large and/or complex programs will produce similarly large/complex constraints, which may take a long time to be solved. The solving cost may be ameliorated by splitting the constraint into independent subproblems that can be analyzed faster and independently. Visser et al. [173] goes a step further with Green, a framework designed to reduce the number of constraint solver/model counter calls by reusing previous results across multiple symbolic execution runs. Green maintains an in-memory database with the constraint solving results. Incoming PCs will be sliced into independent subproblems, which are transformed ("canonicalized") into a normal form to increase reuse opportunities. The canonized subproblems that have never been seen before are solved using off-the-shelf solvers, and their solution is stored on the database; otherwise Green will return the respective stored solution. Jia et al. [85] improves Green in the context of linear integer arithmetic by taking in account *logical implications* between constraints. For example, a solution for the constraint $x > 2$ is also a solution for $x! = 0$, which means that the second formula is redundant and can be discarded if both formulas appear in the same PC. Aquino et al. [6] leverages a canonical matrix representation of linear integer arithmetic constraints to build an efficient search index of solutions for equivalent constraints.

Romano and Engler [147] propose an expression optimizer for bitvector constraints that can learn reduction rules automatically. The optimizer learns new rules in two stages: first, it looks for possibly equivalent constraints using the hash of a sequence of evaluations with specific values. To further improve the performance, the PCs are also split into distinct sets depending on the number/width of the variables. Later, the equivalence map is refined into a set of reduction rules: expressions are simplified and canonicalized, rules that do not reduce the size of the constraint are discarded, and finally equivalence is confirmed using a SMT solver. Lloyd and Sherman [111] simplify path conditions by removing redundant inequalities through polyhedral algorithms, but the approach is limited to linear integer constraints.

Zhang et al. [189], [190] propose a simple approach to reduce the number of calls to constraint solvers, *speculative symbolic execution*. As the name says, speculative symbolic execution delays the feasibility check for the current branch until a (user-defined) number of new clauses is added to the path condition. If the current branch is not satisfiable, the execution backtracks to the last feasible branch, otherwise the execution progresses normally. The authors also propose an optimization to improve performance in scenarios with lots of unfeasible branches: given a reachable branch, if the constraint solver returns UNSAT for one of its children, the other must be satisfiable. Kausler and Sherman [91] propose user-defined backtracking policies for symbolic execution. The rationale is that default backtrack policies, such as a fixed timeout on a solver call, might waste resources due to its strictness. The authors suggest that more flexible policies, like timing out solver calls that exceed the average of previous calls by a certain factor, can lead to improved exploration coverage.

B. Strings

Symbolic execution of user-facing applications, such as web apps, frequently require the ability to reason over string constraints. Unfortunately, the general theory of strings is undecidable [20]; current research efforts focus on discovering theory fragments which may have efficient decision procedures. Early approaches used either automatons to model and decide standard regular expression constructs (e.g. concatenation/union/Kleene star) [46], or encoded string constraints to bitvector formulas [94]. The latter approach can handle mixed integer-string formulas (e.g. using their operator `length` or `indexOf`), but it must decide the size of the final string before attempting to solve it.

The lack of a general theory for strings also poses modeling challenges for symbolic execution: Kausler and Sherman [90] showed that existing string solvers cannot model precisely many of the string operations available in traditional programming languages (e.g. Java or Python). Furthermore, the authors evaluated the modeling cost, performance and accuracy of a selection of string solvers. The experiment used a set of PCs extracted (using dynamic symbolic execution) from the execution of the test suite of string-manipulating applications. Results show that the performance/accuracy/modeling cost varies between solvers, and as such users must consider the characteristics of the expected path conditions before choosing a solver.

Zheng et al. [192] proposed Z3-str, an extension to the Z3 SMT solver capable of solving mixed string constraints. As opposed to previous bitvector-based string solvers, Z3-str treats strings as a primitive datatype, and thus it doesn't need to decide the length of the string before solving. This is made possible by the Z3 plugin system: the core modules (DPLL(T) and congruency closure engines) handle the exploration of the formula and the coordination between the specific theory solvers. Trinh et al. [171] extend the S3 solver [170] with a *progressive* search algorithm for the fragment theory of strings with recursive replacement, length and Kleene Star. The impact of non-termination due to the recursive nature of the operations

is mitigated by pruning the recursion tree when the current branch implies that a shorter subtree is satisfiable. The authors also implemented a variant of conflict-driven clause learning (CDCL) for string constraints.

Both Liang et al. [107] and Abdulla et al. [1], [2] propose new solvers for the fragment theory of unbounded strings with regular language membership and length. The decidability of this fragment is an open problem and thus termination cannot be ensured. However, both authors claim that a practical solver for the fragment can be built. Liang's solver is a proof procedure for a non-deterministic proof calculus, created by combining a off-the-shelf linear integer solver and a EUF (theory of equality plus uninterpreted functions) solver that has been extended to support string and regular language constraints. This allows the solver to be integrated with the DPLL(T) framework of SMT solvers, and not just be a plugin that discharges to other theories. Nevertheless, the procedure was implemented in CVC4 and benchmarked favorably against well-known string solvers; it returns correct solutions very quickly for most of the problems, while it timeouts for a larger share in comparison with Z3-str. Abdulla's approach is very similar: the work introduces a new logic for strings together with a proof calculus based on a set of inference rules. The proof procedure is guaranteed to terminate if the formula is *acyclic*, that is, each variable appears only once in an equality.

C. Non-linear constraints and bitvectors

Non-linear constraints are ubiquitous in software that makes use of complex mathematical operations, such as hybrid control systems and the aerospace domain. In the worst case, non-linear constraint over finite-width values (e.g. IEEE754 floats) can be modeled as bitvector operations: the constraints can be converted (i.e. *bitblasted*) to a set of propositional formulas that match the binary manipulations done in a CPU.

Borges et al. [22] proposes a solver for complex mathematical constraints over floating-point variables that combines metaheuristic search algorithms and interval constraint propagation. Infeasible regions of the domain are removed with interval constraint propagation, which improves the quality of the initial (*seed*) candidate solutions. Experimental results show that the combination can increase the chances of solving a constraint for a small time overhead for the ICP calls. Dinges and Agha utilize a similar approach in [56]: the algorithm, *Concolic Walk*, removes infeasible regions of the domain that are outside of the polyhedra formed by the subset of linear clauses of the constraint. Afterwards, Concolic Walk uses a combination of tabu search algorithm and root-finding numerical methods to find to find solutions inside the polyhedra.

Bagnara et al. introduced FPSE, a Constraint Programming (CP) solver for constraints over floating-point numbers and the four basic arithmetic operations [10]. The performance of CP tools is directly related to how well their filtering algorithms can eliminate infeasible points from the domain. FPSE improves upon previous CP approaches by reformulating existing filtering algorithms to support multiplication and division. the works Constraint Programming approach to *filter* reducing the domain of the variables This application is designed to generate test inputs that execute many paths with floating-point computations.

Fu and Su [63] proposes XSat, a solver that models the solving process as a mathematical optimization problem. XSat builds a function F from the original constraint C such that finding the minimum point of F is equivalent to solving C. Informally, F measures how far a model is from a potential solution. The "representing function" F is translated into a C program, and the optimization process is performed using Markov-Chain Monte Carlo. The tool stops if the optimum is not found after a certain (user-specified) number of iterations. XSat is capable of handling both basic arithmetic and arbitrary floating point functions/rounding modes. Experiments show that the approach is faster than traditional SMT solvers (Z3/MathSAT) and existing metaheuristic approaches [22]. A similar approach using machine learning optimization algorithms is proposed by Li et al. [104].

Tiwari and Lincoln [169] proposes a pseudo decision procedure for nonlinear arithmetic constraints over the reals. The procedure always returns correct results (i.e. it is a decision procedure) if the constraint is a conjunction of *multilinear polynomial equations*, e.g. $xy+4y+1$. It works by decomposing subsets of clauses of the constraint into a matrix form where one of the variables must be one of the eigenvalues of a constant matrix. This rewriting rule is applied recursively for each variable; if the constraint is fully decomposed, it can be decided if a model exists. Otherwise, the procedure returns `unknown`. To support more complex polynomials, the authors implement a few (equi-satisfiable) transformations.

Hadarean et al. [75] investigated the impact of delaying bitblasting when solving bitvector formulas. According to Hadarean, most bitvector solvers attempt to simplify the formula first, and then bitblast it. This *eager* approach discards information that might be used during solving because the word-level structure is lost after preprocessing the formula, thus denying any opportunities for cooperation with solvers for other theories. The authors propose a *lazy* approach: before bitblasting, other algebraic solvers specialized in different fragments of the bitvector theory, are called in sequence (from the least to the most expensive) to analyze the formula. Experiments show that the lazy approach is complementary to the eager one: problems that are easily solved by the lazy solver take a long time with the eager solver, and vice versa.

D. Other theories

Separation logic [145] is a novel, non-classical logic, that gained a lot of popularity in recent years due to its power to reason about heap-manipulating programs. However, separation logic requires specialized decision procedures that aren't easily integrable into existing tools. Piskac et al. [134] attack this problem by proposing a novel fragment of first order logic, called the "logic of graph reachability and stratified sets (GRASS)", to which separation logic formulas can be reduced to in linear time. GRASS can be decided by existing SMT solvers, which allows all the reasoning tasks to be performed

at the solver level and thus leverage existing optimizations and theories.

Daca et al. [50] propose "Arrays-Fold Logic" (AFL), an extension of quantifier-free array integer theory that is able to express *counting* constraints, such as "the number of elements smaller than x is the same in arrays $A1$ and $A2$". Applications of AFL include summarizing loops with branches, checking whether the histogram of an array content matches a discrete distribution, and speeding concolic execution by encoding the statements of a parser as AFL formulas. The decision procedure for AFL encodes the formulas as symbolic counter machines. Decidability for those machines reduce to satisfiability of existential Presburger arithmetic.

Bansal et al. [13] leverages SMT solvers to decide constraints that use domain-specific user defined axioms (i.e. *local theory extensions*). Developers can specify those axioms to "encode" domain-specific properties about theories that aren't supported by the host SMT solver. However, this is done through the use of universal quantifiers; since SMT solvers use incomplete heuristics to guide instantiation, they may never know when to stop enumerating new instances. Furthermore, those heuristics aren't easily customizable; developers must encode their (potentially complete) strategy by preprocessing the input file. Bansal shows that if the axiom/theory extension is local, i.e. you only need to check instances with ground terms that appear in the formula, a decision procedure for this extension can be generated in existing SMT solvers by leveraging E-matching algorithms.

Cristiá and Rossi [49] proposes a new decision procedure for sets and binary relations. According to the authors, existing decision procedures for set theory are incomplete because they rely on a "lossy" encoding to other theories, such as arrays or predicate calculus. The proposed solver, SAT_{br}, implements a complete decision procedure by attempting to reduce the input formula to an equisatisfiable *solved form*. The reduction rules are designed to eliminate problematic expressions, such as constraints over the range of relations, that would result in nontermination. The solver applies the reduction rules nondeterministically until a fixpoint is reached: the result will be either *false* if the original formula is not satisfiable, or its solved form which is guaranteed to be satisfiable.

IV. Path Explosion

Symbolically analyzing all feasible paths cannot scale to large and complex programs, since the number of feasible paths grows exponentially with the increase of branching instructions and can even be infinite in the case of programs with loops or recursion. Therefore, path explosion is a key problem to address in symbolic execution.

A. Heuristics-Guided Path Exploration

Many approaches have been proposed to mitigate path explosion by exploring paths of interest guided by heuristics. One approach was introduced by Li et al. [106] to steer symbolic execution to less traveled paths. The main idea is to exploit "length-n sub-path program spectra" to systematically approximate full path information for guiding path exploration. In particular, it uses frequency distributions of explored length-n sub-paths to prioritize "less traveled" parts of the program to improve test coverage and error detection. This technique recursively update the frequency of traveling through sub-paths of a certain length, and the paths traveled less frequently will be explored first as it is tend to be more possible for a potential problem to exist.

H. Seo and S. Kim used a context-guided search (CGS) strategy to relieve path explosion for concolic execution [156]. CGS looks at preceding branches in execution paths and selects a branch in a new context for the next input. As symbolic execution reached a branch, a *k-context* of this branch will be generate and checked. For instance, in case a symbolic execution reaches a branch *b6* following path *b1, b4, b6*, a 2-context of b6 will be generate as *b4, b6*. CGS will then check whether this 2-context is new or not by checking all 2-context stored in the cache. If it is new, this branch will be select to generate an input following this path, and this 2-context will be stored in cache as well. If it is not new, this branch will be skipped since we assume the corresponding path has been checked before. Meanwhile, in order to handle the situation when k is too small, this framework also calculates the *dominators* to make sure all branches in the k-context is meaningful. In other words, for a certain branch *b*, CGS will calculate the importance of all branches that leads to it. Thus, the k-context of branch *b* will ignore branches that all execution path *must* go through to reach it, only keeping those relevant branches that can generate significantly different paths in its k-context.

A more recent approach developed by Christakis et al. [45] guides dynamic symbolic execution toward unverified program executions. This approach annotates programs to reflect which executions have been verified under which assumptions. Such annotations are used in this approach to guide dynamic symbolic execution, so that tests that lead symbolic execution to verified executions are pruned to avoid exploring parts of the state space, and tests that lead to not yet fully verified properties are prioritized. To be more specific, a program with a property *P*, which is in the shape of assertion, under an assumption *A* will be annotated in the form of *assumed A* where as property *A* is considered true without checking at this point in the program and *assert P verified A* which means when assumption *A* is considered true, assertion *P* should hold. However, since assumption *A* is potentially unsound, dynamic symbolic execution should generate test cases in which *A* is violated to check if assertion *P* still stands true under such condition. With these annotations, this framework is able to guide the symbolic execution in order to prune the redundant tests where *A* still stands.

Coughlin et al. developed an approach [48] to find "validation scopes", which is the minimal parts of code that needs to be checked to verify a property. This technique uses "enforcement windows" which are non-faulty sequence of establish - check - use operations found with symbolic trace interpretation, in order to improve the efficiency of proving properties of interest

with an acceptable number of false alarms using symbolic trace interpretation. Enforcement windows distances are dynamically measured based on two dimensions, namely control and data reasoning, to be used for describing and quantifying the complexity of the validation scopes to provide information needed by designers to observe the spots for improvement.

Zhang et al. [191] introduced regular property guided dynamic symbolic execution to find the program paths satisfying a regular property as soon as possible. They argue that only the paths with specific sequences of events can satisfy the regular property, and the number of such paths is often very small. It is desirable not to explore the irrelevant paths and the relevant paths not satisfying the property.What this application proposes is to explore the off-path-branches, or unchecked branches in current path, along which the paths are most likely to satisfy the property. It combines symbolic execution with model checking on finite state machine (FSM). For each off-path-branches *b*, two FSM models will be extracted: one indicates the path leading to *b*, and one includes the states that could be reached from *b*. The author argued that if the interception of these FSM is empty, the likelihood of a path satisfying the regular property can be considered very low. Thus, such branches should be less prioritized in the following dynamic symbolic execution process. In other words, the symbolic execution would choose other branches along paths that can satisfy the regular property, rather than doing it in a traditional depth first search or breadth first search style.

It is expensive for symbolic execution to systematically analyze functional correctness properties of programs, e.g., using assertions or executable contracts. iProperty [181] facilitates incremental checking of programs based on a property differencing technique. The key insight is that if a property Φ about program p holds at a control point l in p and Φ implies another property Φ', Φ' also holds at l in p. Based on this observation, the algorithm for computing property differences checks for implications between corresponding old and new properties to minimize the set of properties that must be checked after a property change is made and hence reduces the overall cost of checking. Furthermore, for programs with both changes to properties and changes to code that implements the functionality, iProperty uses the property differences in conjunction with code-based change impact analysis techniques, e.g., DiSE [183] to guide checking onto relevant parts of code and properties, thus reducing much redundancy otherwise present in reapplying symbolic execution.

B. Pruning Paths

Since the number of paths and states is potentially large, if we can find a way to prune uninterested states and paths, we can reduce the number of states we actually need to explore and thus relieve the impact of path explosion. One of the useful approaches is using incremental checking. The insight of using incremental checking for path explosion problem is two fold: First, for the same version of a program, we can leverage the state space in fragments e.g. by the depth boundary, and incrementally check different parts of the state space step by step in multiple symbolic execution runs. Second, we can use incremental checking to avoid checking unchanged states and paths as we conduct regression test on different versions of a program.

One of the incremental checking techniques is Directed Incremental Symbolic Execution (DiSE) [132], [183]. DiSE applies symbolic execution and static analysis in synergy to enable more efficient symbolic execution of programs as they evolve.

The static analysis is based on intra-procedural data and control flow dependences. It identifies instructions in the source code that define program variables relevant to changes in the program. Conditional branch instructions that use those variables, or are themselves affected by the changes, are also identified as affected. The information generated by the static analysis is used to direct symbolic execution to explore only the parts of the programs affected by the changes, potentially avoiding a large number of unaffected execution paths. DiSE generates, as output, path conditions only on conditional branch instructions that use variables affected by the change or are otherwise affected by the changes. Based on the development of DiSE, some applications of symbolic execution are developed to increase the efficiency of regression testing [9].

Another approach about how to scale symbolic execution for efficiently analyzing program increments is developed by Makhdoom et al. [119]. This approach focuses more on test cases rather than source code analysis. It patches automated test suites based on code changes. Using the test suite of a previous version, this approach eliminates constraint solving for unchanged code. This technique identifies ranges of paths, each bounded by two concrete tests from the previous test suite, and by exploring them, all paths affected by code can be covered.

Yi et al. investigated postconditioned symbolic execution [184] by pruning paths based on subsumption checking. At each branching location, this approach checks whether the branch is subsumed by previous explorations by checking the summarized previously explored paths by weakest precondition computations. Post-conditioned symbolic execution can identify path suffixes shared by multiple runs and eliminate them during test generation when they are redundant, results in a reduction in the number of explored paths.

Li et al. [105] proved that the subsequent symbolic analysis of two z-equivalent states traverses the same set of paths and gives the same answers to validity and satisfiability queries. Thus, by analyzing one state in a set of z-equivalent states, redundant path exploration can be avoided. They introduced an algorithm that can detect z-equivalent states in a linear scalability, identifying the abstract syntax tree that represents symbolic states in question as unconstrained expressions.

Yang et al. developed memoized symbolic execution (Memoise) [182] to leverage the similarities across successive symbolic execution runs to reduce the total cost by maintaining and updating the state of a symbolic execution run. A trie—an efficient tree-based data structure—is used for a compact representation of the symbolic paths generated during a

symbolic execution run. It is maintained during successive runs in order to reuse of previously computed results of symbolic execution without the need for recomputing them again. Constraint solving is turned off for the paths that were previously explored and the search is guided by the choices recorded in the trie. Moreover, the search is pruned for the paths that are deemed to be no longer of interest for the analysis. Memoise reuses state and path information in previous symbolic execution runs to avoid re-exploring state space that is unchanged or of no interest. Thus, the overall cost of exploring the whole state space is reduced.

C. Merging States or Paths

Some approaches aim to reduce the number of states or paths to explore in general, instead of addressing path explosion problem in certain contexts. One way to reduce the number of states or paths is to merge them. Such a method is introduced by Kuznetsov et al. [98]. This approach first presents a method for statically estimating the impact that each symbolic variable has on solver queries that follow a potential merge point. It merges states only when such merging promises to be advantageous. It then presents another technique for merging states that interacts favorably with search strategies in automated test case generation and bug finding tools. One more recent approach developed by Scheurer et al. [153] reduces number of states by merging symbolic execution branches. This work devises a general lattice-based framework for joining operations and proves soundness of these operations. It has been extensively evaluated with the highly complex TimSort case study and it was demonstrated that significant improvements can be gained.

Jaffar et al. developed an approach [81] to boost concolic testing via interpolation. First, they assume that bug conditions of the program is in the form "if C then bug". Which is to say, if a condition C is satisfied, the path is buggy. Then, the solver will generate an interpolant at each program point along the unsatisfiable path. The interpolant at a given program point can be seen as a formula that succinctly captures the reason of infeasibility of paths at the program point, or in other words, the reason why paths are not buggy. As a result, if the program point is encountered again through a different path such that the interpolant is implied, the new path can be subsumed, because it can be guaranteed to not be buggy.

MultiSE [155] is an approach developed by Sen et al. using symbolic execution on JavaScript programs. This technique merges states incrementally during symbolic execution, without using auxiliary variables. The key idea of MultiSE is to a set of guarded symbolic expressions called a value summary based on an alternative representation of the state. MultiSE does not introduce auxiliary symbolic variables, and updates value summaries incrementally at every assignment instruction.

MergePoint introduced by Avgerinos et al. [7] is a symbolic execution technique that operates on Linux binaries. This approach uses a technique named veritesting, which alternates between static and dynamic symbolic execution to take advantage of each of them. The authors argue that as dynamic symbolic execution (DSE) is a path-based technique for testing purpose, it is suffering from path explosion problem by its natural. On the other hand, however, static symbolic execution (SSE) is treating the program as a whole formula since it is a verification technique to detect potential vulnerabilities. In other words, it only checks the satisfiability of certain formulas in which program problems are encoded as part of it, and since there is no dynamic path exploration, it avoided path explosion problem. Veritesting combines these two symbolic execution techniques basing on the control flow graph of the target program. Starting with DSE process, this technique will not always fork into several sub-states when a symbolic branch is reached. Instead, an analyze will first check if the following program fragment could be statically treated, in which case it will switch to SSE mode and summarize a formula to represent the corresponding program fragment, and switches back to DSE mode for the parts that are difficult to be handled. By treating fragments of the whole program as a "sub-programs" statically, the overall path explosion for DSE is relieved.

D. Handling Loops

J. Strejček and M. Trtík developed an approach on abstracting path conditions [166], which leads the exploration to a certain location while avoiding the path explosion caused by loops. The algorithm is based on computation of loop summaries for loops along acyclic paths, which are called "backbones", that lead to the target location. These backbones are detected by recursively removing the leftmost repeating nodes using loop summaries in the complete path until all cycle states are removed. Given a program and a location within it, the approach produces a nontrivial necessary condition to reach it.

Another approach aims to improve the code coverage in a loop body [92]. This algorithm increases the amount of user-control over symbolic execution of loops by implementing a k-bounded loop unwinding, as the authors argue that loop bounding and search space bounding which are widely used by default could miss important paths and thus test cases generated cannot reach ideal branch coverage in loop bodies. This approach semi-automatically concretizes variables used in a loop by symbolically execute the loop out of context and extract a model for the generated path conditions, following by adding concrete values to these constraints to concretize the variables used in loop bodies or as loop guards. In other words, a complex path constraints inside a loop body will be simplified based on the conditions to cover the branches inside the loop rather than recursively update the path condition by the potential infinite iterations.

S-looper is an approach developed by Xie et al. [179], aiming to deal with loops with multi-paths inside the loop body. For multipath loops, the key challenge for summarizing is that there is a large number of possibilities of a loop traversal for the different execution orders of the paths in it. Typically for a loop condition related to string, it is highly possible that tracking every character in it for loop summarization is necessary but expensive. S-looper automatically summarizes a type of loops related to a string traversal. This approach is to identify patterns of the string based on the branch conditions along each path in

the loop. It then generates loop summary based on this patterns. The summary describes the path conditions of a loop traversal as well as the symbolic values of each variable at the exit of a loop.

Le developed another analysis method named segmented symbolic analysis [100]. This hybrid technique addresses not only challenging problem of loops, but also library calls. It performs symbolic and concrete executions based on demand, similar to a concrete execution. Dynamic executions are performed on the unit tests constructed from the code segments to infer program semantics needed by static analysis.

E. Parallel Exploration

Applying parallel algorithm is another great tool to deal with path explosion. Instead of exploring the whole state space by only one process, we can divide the state space trie into small sub-trees and launch multiple workers to explore them simultaneously. Ranged symbolic execution [159] is such an approach developed by J. H. Siddiqui and S. Khurshid. This approach embodies this insight and uses two test inputs to define a range for a symbolic execution run. It is built base on the idea that the state of a symbolic execution run can be encoded succinctly by a test input. By defining a fixed branch exploration ordering, the symbolic execution is divided into several ranged symbolic execution sub-problems and solved separately.

Synergise [139], [140] extends ranged symbolic execution and introduces two special kinds of ranges – *feasible ranges* and *unexplored ranges*, which lay the foundation of a two-fold synergistic approach: improved distributed symbolic execution and seamless integration with complementary search-based testing tools. A feasible range compactly encodes constraint satisfaction results. Given a feasible range, ranged symbolic execution can be used to quickly populate a constraint database by just building the path conditions for all paths that are within the range – all such paths are feasible by definition – without requiring any additional constraint solving. The constraints results stored in the data-base can then be efficiently re-used when performing symbolic execution in other, unexplored, parts of the program. Unexplored ranges enable symbolic execution to efficiently reuse existing tests, providing a natural integration between any type of test generation tool and symbolic execution. Previously generated tests using other tools, such as random testing tools, can be ordered with respect to the specific search order used by the symbolic execution tool employed to define a set of unexplored ranges, which only contain program paths that none of the existing test covers, and these unexplored ranges can then be efficiently explored by symbolic execution in parallel.

SCORE framework [95] distributes concolic execution [154] so that whole execution paths are generated one by one on distributed nodes in a systematic manner while preventing redundant test case. In concolic testing, a symbolic path formula is extracted on the path traversed during each concrete execution, and further symbolic path formulas are then generated by negating path condition. The concrete executions based on the values generated by solving these symbolic path formulas can traverse different new paths in the program. Rather than exploring these paths sequentially in regular concolic testing, the *SCORE* framework employs distributed nodes to explore these paths in parallel.

V. Compositional Analysis

A. Improving Compositional Symbolic Execution

Since computation is highly demanded by symbolic execution, scalability is one of the major challenges in practice. One of the methods to address this challenge is compositional analysis, a general-purpose methodology that capable of scaling up multiple static analysis and software verification techniques, including symbolic execution. The main idea of compositional analysis is to encode the input-output behavior of each elementary unit (i.e. a method or a procedure) by analyzing them separately. The results of the analysis are stored in a summary for each elementary unit, and by incrementally composing and utilizing these summaries, the whole-program analysis results are obtained. In other words, the preconditions and postconditions of a path are stored in the summary, and for all the later calls of the path that satisfy the preconditions, we can prune applying symbolic execution on the path again, and return the stored post conditions directly instead [108].

Compositional analysis is first proposed by P. Godefroid in 2007 [67] to improve symbolic execution performing along concrete paths [69]. This method is further extended [4] to reach a better scalability, and the latest approach, SMASH [70], employs both "may" and "must" summaries, expressed with logical formulae.

One common problem in compositional symbolic execution techniques that use logical formulas as summaries is that in presence of heap updates, compositional symbolic execution could be less efficient due to the lack of a natural way to compose in presence of heap operations. One approach addressing this problem is developed by J. Rojas and C. S. Pasareanu [146]. This approach is based on partial evaluation (PE) [87]. PE, also known as program specialization, is a well-known technique for automatically specializing a program with respect to some of its inputs. Through this technique, method summaries that consist of path condition and heap constraints for a particular symbolic execution path are obtained, and can be re-executed with reconstructing the heap naturally without keeping explicit representations in them.

Based on this approach, Memoized Reply is introduced by R. Qiu, et al. [141]. This approach summarizes each analyzed method as a memoization tree. The memoization tree captures crucial elements of symbolic execution such as path choices and path conditions for all complete paths. Instead of explicitly encoding the method's outputs, a composition operation is defined to replay the symbolic execution of the methods in different calling methods efficiently, using the memoization trees in a bottom-up fashion. To reduce the number of solver calls, constraint solving is only used to determine which paths in the method summary are feasible at a method call site, and turned off when exploring these paths.

Lin et al. improved compositional symbolic execution by using fine-grained summaries [109]. Different from summarisation on function level used in the conventional compositional symbolic execution, fine-grained summaries are the summarising blocks of code within functions. This change makes summaries more possible to be reused to reduce path constraint solver calls. By reducing the number of solver calls, fine-grained summaries can improve the efficiency of symbolic execution in terms of time cost. The evaluation shows the improvement can be affected by the usage of summaries, which means the more opportunities for summaries to be used, the better improvement we can get.

B. Applying Compositional Symbolic Execution

Besides improving the scalability of symbolic execution, some researchers inspired by the analyze result from compositional symbolic execution itself, and try to take advantage of this information to apply different approaches efficiently. One such approach is Modular And Compositional analysis with KLEE Engine (MACKE) [128] developed by S. Ognawala, et al. This tool enables testers to detect low-level vulnerabilities in a program using symbolic execution in a reasonable amount of time. This solution first performs symbolic execution on individual components in isolation, in order to find out low-level vulnerabilities in these components. The result is reasoned about from a compositional perspective, and the vulnerabilities detected are then scored and report to the user.

DrE is a framework recently introduced by I. Pustogarov, et al. [136] that puts directed compositional symbolic execution into more specific usage. This framework targets software for the popular MSP430 family of micro-controllers. While conventional symbolic execution does not perform good enough on lower-end embedded architecture due to path explosion, directed compositional symbolic execution can be conducted to mitigate this problem and served well as an approach to automatically discover sequences of digital sensor readings that drive the firmware to an adversarially chosen state, especially states of potential vulnerabilities.

VI. Memory modeling

Symbolic execution is playing an increasingly important role in proving memory safety properties or checking correctness of programs manipulating complex data structures, however reasoning on the content of memory brings its own specific challenges and issues. Separation logic [145] has emerged as a major tool to reason about heap manipulation, however there are still restrictions affecting its practical usability that are being addressed in the ongoing research, like its lack of support from current SMT solvers, or the heavy interdependence between how heap properties are expressed and what can be practically verified, while these issues must be tackled within a further exacerbation of the state space explosion due to the large number of possible memory configurations.

This section will focus on the most recent advances in the tools, methodologies and approaches pertaining symbolic execution and memory modeling. Regarding separation logic, since it proved to be useful to tackle also other symbolic execution related problems (e.g. analysing concurrent programs), this section will include only results that are strictly focused on memory modeling, while we refer the reader to section III-D for additional material which is still relevant but more broadly applicable.

A. General advancements

The verification of heap-manipulating programs is a challenging task. Separation logic allows to describe concisely program states that holds in separate regions of the memory, however this resource oriented logic is not supported by current SMT solvers, since the satisfiability of separation logic formulae with inductive predicates and Presburger arithmetic is undecidable.

In [99] the authors identified a fragment of separation logic more expressive than what is used in other current state-of-the-art approaches (e.g. it is not limited to shape-only inductive predicates), and developed an approach to solve satisfiability problems. This fragment is recursive, which may result in the decision procedure taking an infinite amount of time. To handle that, the authors propose an over/under-approximation procedure that unfolds the formula until it finds a model or proves that all leaves are unsatisfiable.

Verification tasks involving data structures that are traversed in various ways (e.g. graphs) is problematic because the recursive predicates generally used to denote their properties effectively restrict the access patterns that can be effectively verified. To overcome this issue in [124] the authors added support to *iterated separating conjunctions* (ISC) in a symbolic execution engine. Iterated separating conjunctions are an alternative way to express properties of a set of heap location, that do not constraint the traversal order, therefore handling an unbounded number of locations at the same time. The proposed approach uses quantifiers over heap locations, however the quantifier instantiation is controlled in a way that maintains a low performances impact. An implementation has been evaluated and showed encouraging results on different verification scenarios.

The verification of code involving the manipulation of data structures is particularly challenging because of the large state space of the data that must be explored. In [65] Geldenhuys et al. focused on bounded exhaustive bug finding, and proposed an approach that increases the scalability of a current state-of-the-art tool (Symbolic PathFinder [116], an extension of Java PathFinder) combining two different techniques aimed at the reduction of the size of the search space. The first is the pre-computation of tight field bounds using the structural invariants of the classes under analysis, which prunes invalid states from the search space. The second is the reduction of the number of partially initialized structures that are considered, by using symmetry-breaking to avoid the generation of isomorphic ones.

Rosner et al. [149] also focused on developing techniques to obtain an early pruning of invalid or redundant data structures, and proposed two mechanisms: bound refinement, which improves over current bounded lazy initialization approach

[65], and introducing auxiliary feasibility checks relying on a SAT solver.

Belt et al. [16] focused on algorithms for the symbolic execution of programs operating with statically-allocated value-based data structures that are found in the development and verification of high assurance applications. Their work focused on Spark [38], a subset of Ada that lacks constructs that are difficult to reason about and includes a notation allowing the specification of pre and post conditions, assertions, and information flow relationships. They provide a formal operational semantic for the symbolic representation of Spark programs, with two popular approaches: a logical representation and a graph-based symbolic representation. Interestingly, the graph-based representation proved to be strictly faster than its logical counterpart.

B. Managing heap input

The symbolic analysis of a program to generate relevant test cases, or to verify certain properties, is particularly challenging when the input is constituted by a dynamic heap structure, as in general the space of possible heap configurations is extremely large and only a small portion of it represents valid input values.

The introduction of the *lazy initialization* approach, presented in [93] by Khurshid et al. in 2003, improved significantly the applicability of symbolic execution in this context, however within the current state of the art there are still methodologies that do not take into account structural properties of the input, performing an extensive exploration that includes isomorphic structures and invalid program executions (thus increasing the analysis cost and raising false alarms) and methodologies that *do* take into account these constraints, but either enumerate in advance all the possible valid structures or perform a consistency check of the whole heap after every new assumption.

Braione et al. [27] proposes a methodology based on domain invariants specifically designed to define constraints over the lazy initialization process, rather than executable predicates over the shape of the data structure, simplifying the verification of the data structure integrity and avoiding other issues, like the fact that the the structure validation procedure itself may trigger the lazy initialization of some fields. This idea is further developed in [28] where the authors explicitly define a language to specify structural constraints of partially initialized data structures, along with a decision procedure that is used to evaluate the constraints in an incremental fashion as the symbolic execution progresses, resulting in a significant scalability improvement over the current state of the art. An evaluation of the resulting tools against multiple experiments over the TSAFE air traffic control application, the Google Closure compiler project, and various programs that manipulates classic recursive data structures, is described in [30], confirming the benefits of the incremental verification.

C. Handling symbolic memory access

The symbolic execution of code containing pointer handling and dereferencing, is a valuable tool to detect software bugs involving invalid memory access operations (e.g.: access to uninitialized or previously freed memory locations), however the symbolic evaluation of operations including pointers is a challenging task, since a symbolic pointer may represent a large number of concrete addresses for which it is necessary to evaluate the validity.

In [148] Romano and Engler developed a dispatcher to resolve symbolic memory access that can be employed on unmodified machine code including also demand-allocated buffers. Their approach defines a separation between the mechanism used to issue a symbolic memory access, and the policy used to resolve it. This allowed to easily support a variety of policies, which is an interesting strength compared with other state-of-the-art solutions, as different policies produce different performance and completeness results, and no single policy can be considered the best. Examples are the *fork on address* policy, which has a high cost but explores all feasible accesses, and the much faster *prioritized concretization* policy, which searches for single a feasible address containing a symbolic value. The library has been evaluated in test case generation and memory fault detection applications, showing an improvement over the current state of the art tools.

In [62] Fromherz et al. describe an enhancement of the Symbolic PathFinder (SPF) for Java bytecode that is capable of performing symbolic execution of programs handling arrays of symbolic length. By using the array theory supported by a variety of SMT solvers, the approach is capable of handling both arrays of primitive types and, thanks to a novel combination with lazy initialization, also of reference types.

D. Advancements in static analysis

Analyzing heap reachability (i.e. checking if an object can be reached through a sequence of pointer dereferences) is a notable static analysis task that is performed to statically check assertions concerning, for example, the lifetime of an object or the correctness of field encapsulation, and is also essential for various other relevant static analysis problems such as escape analysis or taint analysis. In [21] the authors describe a technique for reasoning about heap reachability that is flow-, context-, and path-sensitive, and that provides location materialization. The approach consists in an initial flow-insensitive points-to-analysis that is refined on-demand: the flow-insensitive points-to facts are integrated into a mixed symbolic-explicit representation of the program state, that, together with an algorithm to infer loop invariants for heap reachability queries, is used to perform the analysis, which demonstrated to scale well while maintaining a reasonable precision.

Zhu et al. described in [193] a novel methodology and a tool to prove the linearizability of a concurrent data structure implementations, interpreting it as a property checking problem using separation logic. The tool tries to identify *witness states* of the linearization of the operation with respect to the abstract specification: symbolic execution is used to identify candidate witness states that are then validated, and if every possible path presents a witness state the linearizability is guaranteed.

In [127] the authors focused on a particular class of dangling reference issues that are often found web applications. They observed that these applications often rely on code written in different languages and executed on different physical machines, where parts of the code are generated dynamically; a common configuration (which is the also the reference situation in the paper) involves a server executing PHP code to generate dynamically HTML and JavaScript code that will be then executed on the client side. The lack of cohesiveness between the various components arising in this circumstance may leads to dangling references: certain versions of the generated code may contain variables that have not been appropriately initialized. The authors proposed a static analysis method that uses symbolic execution to build a model that approximates all the possible versions of the generated output, and then uses it to verify that all the necessary initializations are performed in every scenario.

VII. Concurrency

Analyzing concurrent programs is notoriously hard due to their inherent non-determinism. Much research projects have been conducted on applying symbolic execution to analyze concurrent programs. While some are focused on detecting concurrency bugs such as data races and deadlocks, others aim to scale up the analysis of concurrent programs.

A. Detecting Concurrency Bugs

A key question for analysis of concurrent programs is how to deal with data races and deadlocks, which can cause many concurrency bugs. A data race occurs when different threads concurrently access the same memory location, and at least one of these accesses is a write. A deadlock occurs as a thread is holding a lock (L1) and waiting for another one (L2), while another thread is holding L2 and waiting for L1. Thus, none of these threads can continue and the whole process gets stuck. Ideally, data race could be avoided by using proper synchronization between threads, e.g., using concurrent data structures or data-locking mechanisms. However, in practice it is difficult to guarantee data race free for a concurrent program.

Symbolic execution has been applied for predictive analysis [162], which aims at predicting executions that expose races by mutating the schedule order of an execution trace. In particular, Wang et al [175]. and Said et al. [151] proposed symbolic predictive analysis approaches that use symbolic predictive model to precisely capture feasible interleavings and encode the concurrency errors detection problem as an SMT formula. Liu et al. [110] proposed IPA, which allows the schedule mutation to change the location referenced by a shared access, and thus enables more schedule mutations and improves the data race detection. IPA also applies a hybrid symbolic encoding scheme to achieve practical applicability. For operation not supported by the solver, IPA requires all the operands to take the concrete values observed in the seed execution.

Razavi et al. proposed a test generation technique for concurrent programs [144]. It extended sequential concolic execution [33], [68] with predictive analysis [162] for the concurrent setting. The approach alternates between sequential concolic execution and concolic multi-trace analysis (CMTA) to generate tests that will increase structural coverage. Concolic execution is first applied to increase branch coverage on individual threads until coverage is saturated; CMTA is then used to generate new test inputs and thread schedules to cover previously uncovered branches in one of the threads; concolic execution is used again based on these new test inputs and thread schedules to cover more previously uncovered branches. Farzan et al. proposed con2colic testing [58] for performing concolic execution [33], [68] for concurrent programs. Different from traditional concolic execution that is applied for sequential programs, con2colic introduces interference scenario as a representation of a set of interferences among the threads, and extends the traditional sequential concolic execution to model interferences as interference constraints and navigates the space of all interference scenarios in a systematic way. Con2colic testing executes a concurrent program based on a given schedule and stores the observed execution in a forest data structure that keeps track of the already explored interference scenarios, and then decides what new scenario to try next, aiming to cover previously uncovered parts of the program, based on the set of interferences that have already been explored. Deligiannis et al. proposed WHOOP [53] focusing on detecting data races typically in device drivers. Compared to traditional data race detection techniques that are based on happens-before and typically attempt to explore as many thread interleavings as possible (and thus has code coverage and scalability issues), WHOOP uses over-approximation and symbolic pairwise lockset analysis, which scales well.

In addition, several techniques were developed recently to apply symbolic execution on concurrent programs in specific fields. For example, Parallelized Compiled Symbolic Simulation [78] is for verification of cooperative multithreading programs available in the Extended Intermediate Verification Language (XIVL) format. The XIVL extends the SystemC IVL, which has been designed to capture the simulation semantics of SystemC programs, with a small core of OOP features to facilitate the translation of C++ code. Li et al. proposed GKLEE [103], a framework that uses symbolic execution to analyze C++ GPU programs to detect races, deadlocks, as well as performance bugs such as non-coalesced memory accesses, memory bank conflicts, and divergent warps. They introduced a symbolic virtual machine (VM) to model the execution of GPU programs on open inputs.

B. Improving Scalability

Analysis of mutlithreaded programs must reason about possible thread interleavings for each input, in addition to reasoning about program behaviors over all possible inputs. In practice, the number of interleavings grows exponentially with the length of a program's execution, and the interleaving space is too massive to be explored exhaustively [32], [125], thereby exaggerating the well known path explosion problem of symbolic execution. Despite the fact that much research has

been done to relieve the impact of path explosion in general, several studies have been conducted to address such problem in particular for concurrent programs.

Although the number of possible thread interleavings for each input could be large for real-world programs, not all of these interleavings are necessary to check for testing. For example, if there are two threads for a certain program without any data race, it does not matter which thread run first. Kahkonen et al. [89] developed an algorithm that combines dynamic symbolic execution and unfoldings for testing concurrent programs, with a goal of exploring only the interleavings that can lead to a change in software behavior. Unfoldings [120] fights path explosion using a "compression approach" by constructing a symbolic representation of the possible interleavings that is more compact than the full execution tree. Unfoldings naturally capture the causality and conflicts of the events in concurrent programs in a way that allows test cases to be efficiently generated.

Guo et al. [74] developed an assertion guided pruning framework that identifies executions guaranteed not to lead to an error and removes them during symbolic execution. This approach uses a generalized interleaving graph (GIG) to capture the set of all possible executions of a concurrent program. By analyzing GIG and summarizing the reasons why previously explored executions cannot reach an error, redundant executions are pruned in future runs. Guo et al. [73] developed ConciSE which reduces the state space exploration by leveraging change impact information and exploring only the executions affected by code changes between two program versions.

Another idea to limit path explosion for concurrent programs is to symbolically execute relatively small fragments of a program in isolation—this reduces path length, which in turn reduces the potential for path explosion. This idea is first introduced by Bergan et al. as input-covering schedules [17]. This approach implements many optimizations to avoid the inherent combinatorial explosion, like bounding epochs: loops containing synchronization code are analyzed separately and in lockstep for all threads. A further-developed approach [18] is given later on performing symbolic execution of concurrent programs from arbitrary program contexts, rather than starting it at one of a few natural starting points, such as program entry (for whole-program testing) or a function call (for single-threaded unit testing). This approach proposes a way to integrate data flow analysis with symbolic execution. This data flow analysis computes a cheap summary that is used as the starting point for symbolic execution by combining reaching definitions, which summarize the state of memory, with locksets and barrier matching.

VIII. Test Case Generation

The generation of test suites is a major application of symbolic execution currently tackling relevant problems, like the identification of the input values necessary to cover parts of the code that are particularly difficult to reach. To address this kind of problems, in recent years multiple techniques have been developed to improve the usability of symbolic execution.

Batg, introduced by Vorobyov and Krishnan [174], is an application in automatic test generation that combines features of static analysis and bounded symbolic execution. An initial static analysis identifies a list of potential errors on the control flow graph: potentially buggy nodes are identified, and the set of paths leading to these nodes is passed to the "path processor", where symbolic execution is performed. All the paths with satisfiable path constraints are considered as a real bug, and one test case is generated for each of them. Otherwise, these paths are unwound from its "CFG form" where the loops are considered as only one iteration, and the satisfiability of the new generated constraint is checked again. This process is done recursively until a satifiable context is found, which means a test case for the bug can be generated, or the unwinding bound is reached.

Bardin et al. improved the coverage of generated test cases using a different perspectives [14]. They argued that although behaviors related to control are well handled, there is still a high possibility that interesting behaviors related to data are missed in path-oriented criterion. Thus, they defined *label coverage*, a new testing criterion based on *labels* that associated to program instructions, and proved it to be both expressive and efficiently automatable. By appropriately changing the labeling function, multiple coverage criteria can be emulated such as instruction coverage, decision coverage and condition coverage. This new criterion could be used in multiple scenarios to generate test cases of interest with high coverage.

Some recently developed systems are able to not only generate test cases, but also execute them automatically, like the *1-click test executor* technique for Java programs developed by Monpratarnchai et al. [123], which is based on Java Pathfinder and aims to be an easy to use tool with which the user can perform an automated testing of an individual method, class or package. The tool automates the generation of the required driver/stub and then proceeds through a symbolic execution process, which is totally automatic. Then, JUnit test cases will be generated and executed. This technique minimizes the need for input and operation from user, which may require deep understanding of the testing framework as well as the target program via conventional symbolic execution applications.

Siroky et al. developed a technique [161] to verify the constraints on method invocation chains prescribed by an architectural style. It starts by traversing on call graph backward and forward statically. The whole call graph is checked and reduced on a breadth-first search traversing strategy, in order to find all potential paths between the initial method and the final method prescribed by the architectural style. Then, the reduced call graph is used to steer the symbolic execution on these potential paths, and test cases are generated for all feasible paths.

Honfi et al. [80] proposed the use of interactive graphical visualization of symbolic execution to help identifying issues with the currently generated test cases for non-trivial programs, and developed this idea in SEVIZ *(Symbolic Execution VIsualiZer)*. The fundamental motivation of this tool is to level the gap between research settings and actual application when

transferring symbolic execution to industrial applications from ideal academic environments. This tool visualizes symbolic execution in a form of symbolic execution trees in order to provide a direct information to enhance the test case generation. SEVIZ applies three different analyzing models, namely execution identification, loop identification and path condition identification to simplify the potentially over-detailed reports and logs which are common in research settings, thus reducing the following workload for discovering problems in data collected from symbolic execution in industrial scenarios.

A. Improving Test Case Generation Efficiency

One important approach to improve the test case generation efficiency in symbolic execution is to combine it with conventional concrete execution. Such approach is named "concolic execution" and is widely used. Concolic execution first executes programs with concrete inputs and records symbolic path constraints on the fly, and then uses the collected path constraints to generate input values that can go through all the un-executed paths. In this way, it combines symbolic execution with the benefits of fast concrete execution, with the possibility of generating new concrete values, triggered by symbolic constraints, in order to explore additional, potentially rare, program behaviors.

You et al. developed a technique [186] that uses the Observable Modified Condition/Decision Coverage (OMC/DC) [177] criterion to generate test suites. Dynamic symbolic execution is employed as the method to ensure generating high quality test suits with high criterion coverage. OMC/DC is an improved coverage criterion based on Modified Condition/Decision Coverage (MC/DC), which is a condition-based criterion that is widely used in safety-critical systems. In order to reach a high OMC/DC score, conditions are tagged to analyze the observability. Such tags are used in the adopted concolic execution in a novel approach for generating test suits incrementally, aiming to cover all possible outputs propagated from conditions that with expected observability so that the test suits can meet the demanded OMC/DC coverage.

Following the guiding principle of RANDOOP [130], which relies on feedback-directed random input generation, Garg et al. developed a technique using improved automatic unit test generation for programs written in C/C++ [64]. The main idea is to improve the coverage obtained by feedback-directed random test generation methods. It employs concolic execution on the generated test drivers, and uses non-linear solvers in a lazy manner to generate new test inputs for programs with numeric computations. The main novelty of this approach is that it conducts analysis on unsatisfied cores returned by SMT solvers, checking whether a branch can be reached with another input, or can be used to tell whether certain test suits cannot reach a target branch.

Luckow et al. introduced JDart, a concolic execution extension for Java Pathfinder [55], [113]. A distinguishing feature of JDart is its modular architecture: the main component that performs dynamic exploration communicating with a component that efficiently interfaces with constraint solvers to construct constraints. These components can be extended or modified to support multiple constraint solvers or different exploration strategies.

B. Specifying Testing Methods

Beside all the above applications designed for a general test case generation purpose, there is also progress on using symbolic execution on a specific kind of testing method.

1) Patch Testing: Software evolves more and more frequently in recent years. In some cases, the update could happen in days via patches and is delivered to a mass number of users. Thus, how to improve the efficiency of testing for evolving software draws attention of many researchers. Techniques using symbolic execution for regression testing and patch testing have been developed in recent years.

One such technique developed by Jamrozik et al. [82] is augmented dynamic symbolic execution (ADSE). It is designed to use dynamic symbolic execution for better regression testing. ADSE augments path conditions with additional condition that enforce target criteria such as boundary or mutation adequacy, or logical coverage criteria.

Qi et al. [138] developed a partitioning method of program paths based on program output, to produce a *semantic signature* that is more concise than a complete enumeration of all possible execution paths by conveying enough information to reason effectively about changes between different program versions. The output of the symbolic execution is analyzed on-the-fly, grouping together paths that with the same symbolic output. Then a *semantic signature* of the program is generated. The signature can be further used to detect changes between different versions with higher efficiency, which is frequently done in regression testing.

Marinescu and Cadar developed ZESTI (Zero-Effort Symbolic Test Improvement) [51], an approach to improve regression testing. While regression test suite is usually a limited number of input-output pairs, this approach takes advantage of symbolic execution to check all possible inputs on the same path towards a certain operation in the program. Meanwhile, regression test suits are also used as guidance for symbolic execution. Theses slightly divergent but potentially buggy paths are explored by this guided symbolic execution, detecting more potential bugs in the program without any extra effort from programmer or testing team.

Shadow Symbolic Execution [36] is is a technique developed by Cadar and Palikareva, named after the way it runs: in one symbolic execution instance, two different program versions will be run together at the high level, while the information of the old version is gathered and used to "shadow" the new one. By doing so, the changes between versions are effectively detected by the precise dynamic value information gathered, and the symbolic execution are driven based on these differences. In other words, symbolic execution on the new version will generate test inputs addressing actual changes between two versions, instead of only generating test cases that make both versions behave identically.

FSX, recently developed by Yoshida et al. [185], is a technique designed for incremental unit test generation for C/C++ programs. The technique uses a diagnostics engine to guide and refine test-drivers and symbolic execution, and dynamicly analyze the dependency with Reduced Ordered Binary Decision Diagrams (ROBDDs) [31]. The concolic execution process is conducted incrementally: it starts with concrete input values as a minimal test suite, and the diagnostic information will be obtained during the symbolic execution. Based on this information, only variables that are relevant are made symbolic in the test driver.

Le and Pattison designed a framework named Hydrogen for patch verification [101]. This technique performs symbolic analysis on multiversion interprocedural control flow graph (MVICFG). This graph represents the differences among the various versions of the program, and is the central element used to verify relevant aspects of a patch, namely whether it correctly fixed an existing bug without introducing new bugs, and whether it can be applied on all software releases impacted by the bug.

2) Other Testing Methods: There are also techniques aiming to optimize test case generation for other testing methods:

- Data Flow Testing: Data flow testing (DFT) focuses on the flow of data through a program. Su et al. introduced a hybrid DFT framework [167]. The core of this framework, which is based on dynamic symbolic execution (DSE), is improved with a novel guided path search strategy. The reachability checking technique origins from the DFT problem in software model checking, and transformed into DSE-based approach. First, program will be statically analyzed to find def-use pairs. Then, a dynamic symbolic execution run will be applied on program, generating test inputs that covers as many pairs as possible with in a time boundary. All uncovered pairs will be checked using CEGAR based approach, checking feasibility for each of them. This result will then be used to run dynamic symbolic execution with a higher time boundary. Feasible pairs will be covered as many as possible through multiple cycles, with the time boundary increasing for each cycle. With this technique, the guided path search strategy is implemented to cover data-flow test objectives as quickly as possible in order to mitigate the path explosion problem.
- Load Testing: Different from normal testing case generation, it focuses on generating test suites for load test. The approach introduced by Zhang, Chen and Wang [188] addresses the limited effectiveness of load test generation on large software pipeline by performing the test generation compositionally. Performance of system components are analyzed separately with different symbolic execution based techniques. The performance report from these analyses are summarized and gathered, and based on another analysis across these summaries, load tests for the whole system are generated. This approach is currently capable to be applied to any system with a structure of software pipeline.
- Mutation Testing: Mutation testing is a white box testing method, in which certain statements are slightly changed ("mutated") and applied on test cases. The purpose of mutation testing is to check the robust of test cases and make sure it can detect the change in software behavior due to the mutant code. A common problem in mutation testing is that due to the equivalent mutants introduced, the mutant score may become worse along with the development of system and creation of new mutants. In other words, the mutant score may become not as reliable as the mutation testing continues. To address this problem, Nequivack is developed by Holling et al. [79] to show how to establish non-equivalence or "don't know" among mutants. "Don't know" mutations indicates those mutants which are not equivalent to the original program. This technique calculates the number of this kind of mutants by combining static analysis and symbolic execution, and mark the impossible verification it detected as one such "don't know" mutations. The lower the number of these mutations, the more reliable the mutation score.

C. Non-traditional Programs

Conventional symbolic and concolic testing has been used to provide high coverage of paths in statically typed languages. Meanwhile, we found that applications using symbolic execution on other languages are facing unique problems. In order to address these problems, more and more approaches have been developed focusing on specific kinds of applications, e.g. web applications.

1) Web Applications with JavaScript: Symbolic execution has to face specific challenges in the domain of JavaScript based Web applications which are quite different from traditional programs. Web applications are usually event driven, user interactive, and string intensive. Also, unlike traditional programming languages, JavaScript is a dynamic, untyped and functional language. To address these challenges, some techniques have been developed to apply symbolic execution on web applications using JavaScript.

Li et al. introduced SymJS, a comprehensive technique for automatic testing of client-side JavaScript Web applications [102]. This framework is composed by two major parts: a symbolic execution engine designed for JavaScript language and an automatic event explorer tool for Web pages. Web events are discovered through the symbolic execution of the associated JavaScript. After this process, dynamic feedback from all these execution is refined, and SymJS generates test cases with high coverage based on it. The whole process can be performed automatically, without the need of users supervision and intervention.

Another technique using type-aware concolic testing of JavaScript programs is developed by Dhok et al. [54]. This technique is developed based on the observation that the number of inputs generated for JavaScript programs can increase dramatically, and one of the main reason is that a naive concolic testing to these programs are type-agnostic. The implementation of this technique seems simple, which is introducing a type-awareness approach to conventional concolic testing, but the

result can effectively reduce the explosion in number of testing inputs for JavaScript programs.

ExpoSE [112] is a more recently published technique by Loring et al. This technique is designed to generate test cases with a high path-coverage score for JavaScript programs. While being a tool particularly designed for JavaScript programs, the largest novelty of this technique is that it addresses one typical problems in symbolic executing JavaScript programs, namely the difficulties of applying SMT solvers on regular expressions that are widely used in JavaScript codes. The authors give a solution that nested capture groups and back-references are formalized in JavaScript's flavor with restrict , so that the regular expressions are encoded into logical constraints which can be solved by SMT solver.

2) Smart-phone Applications: There is a growing need for automated testing techniques aimed at Android apps. The development of test case generation using symbolic or concolic testing approaches for Android apps are widely developed. Some of the platforms have been so well developed that can be used in industrial scenarios, such as CREST-BV and KLEE [95]. Applying symbolic execution tools to generate test cases for Android apps is challenging. Generally speaking, Android apps are commonly developed in Java. However, the Android platform is different from the "traditional" JDKs on PCs: Android apps are event-driven and susceptible to path-divergence due to their reliance on an application development framework [122].

Different from a common software, behaviors from users are much more frequent and important in Android applications. Thus, in order to test these applications automatically, users' behaviors must be simulated accurately. Mirzaei et al. introduced this problem and developed a model of Android libraries in Java Pathfinder (JPF) to enable execution of Android apps [122]. Four years later, they implemented a tool based on their research called SIG-Droid [121], which leverages the knowledge of Android's ADF specification to automatically extract two models from source code of an app: the Behavior Model (BM), and the Interface Model (IM). These two models are used to generate event sequences equivalent to simulate a sequence of user's behavior.

An algorithm called "ACTEVE" is introduced by Anand et al. [5], aiming at using concolic testing on Android apps. This algorithm focuses on tapping events to drive the the exploration. Symbolic variables are used as the co-ordinates of the tapping event, with the frame layout used as the constraints. Also, this technique used a concolic algorithm: In the case of tap events, whenever a concrete tap event is input, this instrumentation creates a fresh symbolic tap event and propagates it alongside the concrete event. As the concrete event flows through the SDK and the app, the instrumentation tracks a constraint on the corresponding symbolic event which effectively identifies all concrete events that are handled in the same manner. In order to mitigate path explosion problem, this framework also introduced analysis of subsumption identification, and does not extend paths leading to a "read-only event".

Jensen et al. [84] addressed this problem from a different angle. Instead of applying an approach that generates test suites with a high coverage score, this technique is particularly designed to reach a certain line-of-interest in the Android application code. This two-phase technique will first discover and summarize the event handlers of the application by conducting a concolic execution, and in its second phase, event sequences will be generated backward from the target using these summaries. Thus, an event sequence from the entry of the application towards the given target line is generated automatically and can be used for testing purposes.

3) Database Management Systems: Although software systems that interact with a database are very common in practice, fully automated generation of test cases for database itself is difficult. Traditional symbolic execution approaches [57] are usually not designed to test stored procedures themselves. This kind of techniques consider databases as external and are always facing the challenges introduced by the multi-lingual nature from applications, and thus limited. To address this problem, an approach using dynamic symbolic execution for stored procedures in database management systems is developed by Mahmood et al. [118]. This technique considers database as an internal element for symbolic execution by treating values in database table as symbolic. Data in tables are treated as symbolic variables, while data types, expression operations, function calls and database constraints (check, unique, keys, etc.) are extracted and used to build a "plan tree" model, which is later processed using symbolic execution. Then, with the help of SMT solver, this framework is capable to generate test cases that executes different paths and discover input values leading to schema constraint violations or user defined exceptions.

D. Optimization

Symbolic execution is capable to generate a large number of test cases in order to meet the maximum possible coverage. However, some researchers argue that not all of the test cases are equally important, while some of them can even be considered "unnecessary". They have developed different algorithm and technique to optimize the test cases generated by removing unnecessary test cases or prioritizing them.

"Don't Care Analysis" [126] is such a technique introduced by Nguyen et al. in which they argued that symbolic execution engine often generates test cases which are not easy to be interpreted by human, and such test cases could be and should be removed from the collection. "Don't care analysis" identifies assignment statements that can be safely removed from the test cases without affecting the overall code coverage. This technique is based on binary and delta-debugging search, and it is the first fully automatic approach that reduces the sizes of test cases generated using symbolic execution.

Rapos and Dingel introduced another interesting technique [143] that uses fuzzy logic and symbolic execution to prioritize UML-RT test cases, in order to prevent unnecessary generation of tests. The approach follows the pattern of fuzzy logic control systems to prioritize each test case in a UML-RT test suite, based on natural language rules about testing priority. A similar prioritizing technique, Document-Assisted Symbolic Execution

(DASE) [178] is introduced by Wong et al. the same year. This is an approach to enhance the effectiveness of symbolic execution for automatic test generation and bug detection. DASE extracts input constraints from documents and uses them as a "filter" to favor execution paths that execute the core functionality of the program.

IX. Security

A. Hybrid techniques

Fuzzing is a testing approach that consists of feeding large amounts of random inputs to the target program in a attempt to reveal bugs. Recent works attempt to mitigate the weak points of symbolic execution and fuzzing by combining them in a hybrid technique.

Haller et al. [76] uses a combination of fuzzing and taint analysis to guide symbolic execution to areas of the program that may contain buffer overflows. The technique, Dowser, works in five steps: 1) array accesses in loops are collected and ranked according to their complexity; 2) identify through taint analysis which parts of the input (obtained from existing tests or fuzzing) exercise the array accesses; 3) Learn what branches have a higher probability of leading to unique pointer dereferences (*access patterns*) by performing symbolic execution with a small symbolic fragment of the relevant section of the input; 4) Perform symbolic execution with the entire relevant input fragment made symbolic while guiding the exploration towards the branches found in the previous step; 5) use off-the-shelf tools (e.g. AddressSanitizer [157]) with the generated inputs to check the existence of buffer overflows.

Stephens et al. follows the opposite approach with Driller [164]: symbolic execution is used only to help the fuzzing engine to get 'unstuck'. The underlying fuzzing engine, AFL [1], keeps track of which basic blocks of the program have been visited so far. Once AFL cannot find inputs that cover new basic blocks after a certain time, Driller performs symbolic execution targeting the frontiers of the explored area of the program so far (called *compartiments* in the work). The generated input is then sent to AFL, which will continue the fuzzing process by generating similar inputs. Driller is one of the components of the Mechanical Phish platform, which achieved third place in the DARPA Cyber Grand Challenge competition.

B. Embedded systems

Applications running in embedded devices are potentially vulnerable to physical attacks. For example, attackers can introduce faults in the program execution by manipulating hardware components (e.g. shining a laser at the memory banks), thus avoiding the execution of security routines. Potet et al. [135] proposes Lazart, an approach to evaluate the robustness of applications against fault injection vulnerabilities. Given a target statement, Lazart identifies basic blocks that may impact the reachability of the target. These basic blocks are potential targets for fault injection attacks. Lazart then builds a high-order mutant program [86] that encodes all possible sets of faults. The final step uses concolic execution (through KLEE) to find all paths that lead to the target statement; assertions are used to ensure realistic scenarios (e.g. make sure that only invalid inputs are considered). The number of valid paths found + the number of faults required for each path can be used as a measure of the robustness of the program.

Davidson et al. present FIE [52], a KLEE-based concolic execution tool for 16-bit microprocessor firmwares. Since different microprocessors may have distinct characteristics such as memory layouts and interrupt handlers, FIE allows the user to specify these settings in a modular way. FIE also introduces two optimizations intended to help the exploration get out of deep loops. The first one is *state pruning*, where FIE removes states that match already explored states from the queue. To speed up the matching, FIE stores the diffs of all states for each visited value of the program counter. The second optimization is *memory smudging*: loop counters are replaced by symbolic variables after a certain number of iterations, allowing the analysis to escape the loop at the cost of precision. FIE was evaluated in a set of benchmarks consisting of a corpus of programs taken from github, plus a few well-known firmwares. It was able to reach a high coverage (90%) for small and medium programs (number of executable instructions < 500), but it covered a lot less in large programs ($< 25\%$).

C. Underconstrained Symbolic Execution

Ramos and Engler [142] propose to perform symbolic execution in individual functions of the program (*under-constrained* symbolic execution) in a attempt to avoid the burden of state-space explosion. Although underconstrained symbolic execution is imprecise, the authors claim that it is still powerful enough to be useful in certain scenarios, such as regression testing of patches. To evaluate the technique, the authors developed UC-KLEE, an extension of KLEE for underconstrained symex. UC-KLEE leverages lazy initialization to handle pointers, and allows the user to specify preconditions to silence false positives encountered by the tool. UC-KLEE was evaluated in two scenarios:

1) checking if patches introduced new crashes by generating test harnesses. If the patched version crashes but the original version don't, an error is reported. Path pruning is used to avoid wasting time in uninteresting paths; for example, paths that lead to an already found error (based on program counter) are pruned.
2) Checking all paths of the function for memory leaks and uninitialized data.

The evaluation was performed on large open-source C programs, such as the Linux Kernel and OpenSSH. Results are positive; UC-KLEE found almost 80 bugs on the subjects, although the false positive rate was considerable.

Jana et al. [83] leverages underconstrained symbolic execution to identify incorrect error handling in C programs. The tool, EPEX, uses an error specification for the API call under test + symbolic execution to look for paths in which the API function may return an error. EPEX then checks if the error is handled correctly, i.e. (from the paper) (i) pushed

[1] http://lcamtuf.coredump.cx/afl/

the error upstream by returning a correct error value from the caller function, (ii) stopped the program execution with a non-zero error code, or (iii) logged the error by calling a program-specific logging function. To reduce false positives, EPEX doesn't report an error if all paths ignore the error value. EPEX was evaluated on SSL/TLS libraries, like OpenSSL and GnuTLS, and applications that use them. It was able to find real bugs in most of them, although the false positive rate was higher for the applications due to the complexity and higher number of configuration parameters (i.e. flags to ignore errors).

Yun et al. [187] use a similar idea, *relaxed symbolic execution*, to learn possibly correct API usage patterns directly from the source code. Relaxed symbolic execution is strictly intra-procedural and only unrolls loops a single time, while underconstrained symbolic execution is interprocedural and uses k-bounded loop unrolling. APISan, the presented tool, uses relaxed symbolic execution to extract a few representative paths, which are analyzed to extract *semantic beliefs* about usage patterns API; the probability of a pattern being correct is associated with its frequency in the code. The authors evaluated the effectiveness of APISan in large open-source applications (Linux Kernel, OpenSSL); APISan found 76 unkown bugs (69 confirmed by the developers at the time of writing).

D. Ad-hoc solutions and enhancements

Wang et al. [176] proposes MetaSymploit, a symbolic execution-based approach to automatically generate signatures of attack scripts written in the MetaSploit framework [2] to be used in Intrusion Detection Systems. Given a MetaSploit attack script, MetaSymploit generates a signature in three steps: 1) extract symbolic behavior descriptions of the script, like sequences of api calls and attack payloads, through symbolic execution; 2) Consolidate the information obtained in the previous step into rules that can be used by Intrusion Detection Systems (IDS) to detect the attack. This happens in three stages: extract constant patterns (payload length/offset/content), remove redundancies/benign data (e.g. "Content-Type:text/html"), and derive the context from the sequence of API calls. MetaSymploit seems to be successful in generating signatures for a large amount of real-world MetaSploit scripts; most of the failures can be attributed to limitations in the symbolic execution engine, such as api calls without a model or non-termination due to loops.

Chau et al. [40] propose to test X.509 PKI libraries for non-compliance through the use of symbolic execution. Validating X.509 certificates is a complex task due to the complexity of the specification; applying symbolic execution directly is inviable. The authors propose instead to use *SymCerts*, certificates that mix concrete and symbolic data, to verify the libraries compositionally. This can be done by splitting the certificate fields into independent logical sets that can checked in isolation. For each set, a chain of SymCerts is created with all fields contained in the set made symbolic, and all others made concrete. The chain of SymCerts is used as input for the libraries under test, and the resulting path conditions are used to perform cross-validation. The approach was able to find 48 instances of non-compliance in open-source SSL/TLS libraries.

Pasareanu et al. [131] propose to combine symbolic execution and Max-SAT solvers to quantify the information leakage of a program after a fixed number of executions. Programs that leak large amounts of information through side-channels, such as execution time or memory consumption, may be vulnerable to attacks. Intuitively, the leakage of a sequence of executions is proportional to the number of distinct *observables* (side-channel values observed); an upper bound can be obtained through well-known information-theoretic measures, such as channel capacity. The proposed approach computes the maximum leakage across k executions in three steps: 1) collect all PCs and their respective observables through symbolic execution. 2) find the k-sequence of executions with most distinct observables through Max-SAT. For each distinct observable sequence, a Max-SAT clause is created containing the disjunction of all PCs that match the sequence. 3) Compute the channel capacity of the sequence. The authors also propose a greedy version of their algorithm that analyses each execution individually; the greedy algorithm is more efficient, but it may not return the actual leakage upper bound.

Xu et al. [180] propose CryptoHunt, a tool that leverages symbolic execution to identify cryptographic functions in obfuscated binaries. CryptoHunt works in three steps: first, the tool extracts a trace of the binary under analysis and attempts to identify possible loop bodies. Next, CryptoHunt encodes the relation between the loop input/output variables as a bit-accurate boolean formula, which preserves the semantics of the non-obfuscated (original) program. This is done by performing symbolic execution on backward slices starting from the loop output variables. Finally, the tool attempts to prove the equivalence, using SMT solvers, between the formulas extracted from the binary under test and a set of reference formulas for known cryptographic algorithms taken from open-source libraries. Since there are multiple ways of matching the input/output variables for each formula, the authors propose a mapping algorithm based on the matrix mapping problem capable of filtering unfeasible mappings quickly.

Stoenescu et al. [165] present RIVER, a new binary analysis framework. RIVER translates x86 instructions to a intermediate representation that allows reversing (i.e. "undoing") operations. Symbolic execution is implemented on top of this reversible IR, which reduces memory costs by undoing operations instead of backtracking. However, the current implementation of the IL results in a 6x increase in size compared to the original code; the authors claim that it can be reduced to 2x with some optimizations. System calls and external libraries are handled by snapshotting the state before the function call, since there's no way to reverse the side effects in those cases. RIVER is currently used internally at BitDefender; no public releases were available at the time of writing.

Hasabnis and Sekar [77] propose EISSEC, a tool for extracting mappings between a compiler's intermediate representation

[2]https://www.metasploit.com

(IR) and architecture-specific assembly instructions. Modern compilers use architecture-specific sets of rules called *machine descriptions* (MDs) to drive a generic code generator. MDs contain a mix of instruction pattern rules and auxiliary code, which may be many times larger than the rules description; manually extracting IR to assembly mappings from the MD is very costly. EISSEC simplifies this task by computing all solutions to all possible paths on the code generator (and the target MD) through concolic execution. The typical structure of code generators (no complex loops/pointer usage, finite input domain) makes this task feasible. Instead of traditional SAT/SMT solvers, EISSEC uses a CLP solver to check the satisfiability and efficiently enumerate all possible solutions of constraints.

X. Probabilistic Symbolic Execution

Probabilistic symbolic execution (PSE) is an extension of symbolic execution aiming at computing the probability of a specific event to occur during a program execution, assuming the program inputs follow a given probabilistic distribution. The input distribution allows both developer assumptions and data from the real-world to be incorporated in the analysis, thus tailoring the analysis to specific usage profiles. PSE has many potential applications, e.g., it can support debugging by allowing a quantitative ranking of detected program errors [129], for analyzing the control software of an autonomous agent interacting with an uncertain external environment, for computing software reliability or expected execution time [41], or for quantitative information flow analysis [133] for security applications, including preliminary attempts at analyzing encryption routines [25].

PSE has been firstly proposed in [66]. The basic intuition of the authors was to enrich the transitions in a symbolic execution tree with a *count* of the number of solutions, or models, of each path condition. The approach was limited to linear integer constraints, which allow for efficient model counting techniques based on Barvinok's algorithm [15]. The ability to count the number of solutions reaching each node of the symbolic execution tree enabled not only to rank different error paths, but also to steer the symbolic execution driver towards the most unlikely paths, under the intuition these are often less likely to be stressed during quality assurance phases. A set of techniques, including normalization, caching, and relaxation of complex constraints (which entails the computation of sound bounds on the number of approximate solutions, instead of exact counts) have been proposed by the authors to cope with the complexity of a trivial application of model counting. All the input variables in this work are assumed uniformly distributed within a specific interval.

In [60], PSE has been extended to support arbitrary input profiles, nondeterminism, and a more efficient handling of exact counting. This works still focused on linear integer constraints. Input profiles were to be specified as a map from an arbitrary partition of the input domain to the expected probability of each subset in such partition to be observed during execution; elements within the same subset were assumed to be equally probable. For example, the input domain of an integer variable can be split into multiple ranges and each such ranges can be assigned a numerical probability to describe an histogram distribution. This allowed to tailor PSE analyses to specific usage scenarios, formally computing the probability of a target event to occur conditioned to the program being used as specified. To support nondeterminism, the internal nodes of the symbolic execution tree have been classified in *conditional* and *nondeterministic* choices. For conditional nodes, the probability of each successor can be computed using model counting techniques (enhanced to take into account the usage profile). Instead, nondeterministic nodes allowed only to reason in terms of best and worst case scenarios, by selecting either the successor node leading to the largest or the smallest probability of exposing the target event during successive execution steps, respectively. The result of the analysis in presence of nondeterminism is thus no longer a single probability value but a range of probabilities, accompanied by schedules driving the program towards reaching the target event for the maximum and the minimum probabilities in the range, respectively. This analysis is reminiscent of dynamic programming techniques used for the analysis and the synthesis of optimal schedules for Markov decision processes [11], but can be applied directly on the implemented code artifacts instead of requiring abstract modeling.

Statistical Symbolic execution. To deal with the scalability issues induced by model counting and nondeterminism, two different approaches have been proposed in [61] and [115], respectively.

In [61], the authors introduced the concept of *statistical symbolic execution*, where the accuracy of the result is traded for scalability by allowing the analysis to only sample a finite number of execution traces and to infer the probability of the target event using statistical methods on those samples. Because an unbiased sampling of the execution paths requires to compute the satisfaction probabilities along the branches traversed by the path, statistical symbolic execution can enhance the statistical estimators with exact information about the parts of the symbolic execution tree traversed during sampling. This distinguishes the approach from classic statistical model checking because the analysis draw conclusions combining a partial exact analysis with a statistical process, guaranteeing a faster converges to the accuracy and confidence goals set by the user. Furthermore, because each path that has already been samples once is exactly analyzed, it can be safely excluded from subsequent sampling rounds. This observation allows to define an iterative sampling procedure that samples new paths only from the parts of the symbolic execution tree that have not been explored yet, increasing the chances of selecting also lower probability paths, which is the main limitation of purely statistical approaches [150].

In [115], the authors propose an approximate solution for the analysis of programs exhibiting nondeterministic behaviors, such as multi-threaded programs. To deal with the exponential explosion in the number of possible interleaving, the authors propose a to use a Q-learning [88] algorithm adapted from

reinforcement learning. Q-learning aims at establishing a near optimal trade-off between exploration of the possible schedules and exploitation of the information gathered during such process. When the scheduler is presented with a choice between two competing actions, it uses the knowledge gathered before to bias the decision towards the action most likely to lead to the optimal schedules, i.e., the schedule that maximize (or minimize) the probability of satisfying a given property. Similarly to [61], the use of statistical methods allow to handle large programs, but can only provide probabilistic convergence guarantees, i.e., given enough time, the optimal schedule will be found with probability 1.

Beyond linear integer arithmetics. Probabilistic symbolic execution relies on model counting procedures to compute the probability of satisfying the path condition leading to the occurrence of a target event. While efficient model counting procedures based on Barvinok's algorithm [15] are available for linear integer arithmetics (Latte [12],Barvinok [172]), other domains may require more complex counting procedures if not the (usually unfeasible) enumeration of all models.

Floating-point constraints have been studied in [23], [24] for a direct application to PSE. The authors use Monte Carlo techniques to compute the expected probability of generating an value that satisfy a given constraint from a prescribed input distribution. To increase the precision of the estimation, and to increase the reusability of previous (partial) solutions, the authors introduced a divide-and-conquer strategy where complex constraints are partitioned into independent sub-constraints that can be analyzed independently. The results are then composed by propagating both the estimate and a measure of convergence (variance) through the composition rules. Furthermore, interval constraints propagation techniques [72] are used to reduce the uncertainty about the estimate by focusing the Monte Carlo sampling only towards the regions of the input domain that may contain models of a constraints, pruning out the rest of the domain. However, Monte Carlo approaches have usually limited effectiveness when the number of models of a constraint is significantly smaller than the size of the domain, because the randomized sampling is unlikely to hit any such models out of chance. In this case, more complex techniques can be used [25], which rely on bit-blasting to reduce the estimation problem to counting the models of a SAT constraint (#SAT).

String constraints and counting procedures for their models have received a significant attention in recent years. In [117], the authors propose a method based on ordinary generating function for the approximate counting of the models of string constraints. The supported constraints include predicates stating a string is matched by a regular expression, equality between strings, substring tests, and a limited set of predicates on the length of a string. The procedure computes approximate counts in the form of intervals that soundly enclose the exact count; the generating functions computing the intervals' bounds are obtained directly from the syntactic structure of the string constraint. In [8], the authors propose an automata-based model counting procedure for string constraints. The procedure is composed of two steps. First the constraint is encoded in an automata that accepts all only the strings satisfying a given constraint. Then, established methods based on generating functions are used to count the number of accepting paths of the automata, up to a maximum string length. This procedure allows to obtain exact counts for linear string constraints. To increase the expressiveness of the constraint language, the authors also support a set of non-linear string constraints, for which an overapproximating automata is built, in turn producing an upper bound on the number of models.

Mixed theories and other constraints represent the combination of constraints whose satisfaction is to be evaluated over multiple theories. For example, a string may be required to be matched by a certain regular expression, while its length must satisfy a nonlinear integer constraint. In these cases, more general #SMT solvers may be needed. As exact counting in such setting may result in complexity close to the enumeration of all the models of a constraint, approximate solution have been proposed, as in [44], where established hash-based approximate counting techniques from the realm of #SAT [39], [71] are adapted for #SMT problems. This area has the broad potential for supporting model counting for the broader set of constraints currently handled by SMT solvers, though still suffer of limited scalability.

The definition of logics for expressing properties of data structures and the corresponding algorithms for counting the number of their models is also an open problem, with only limited solutions produced so far in literature [59].

XI. Tools Support

- Symbolic PathFinder [116] (SPF): Built as an extension of Java PathFinder, a well-known software model checker, SPF performs a non-standard interpretation of JVM bytecodes to enable static symbolic execution. SPF supports multithreaded programs by leveraging the underlying model checking framework to explore all possible thread interleavings. The tool is open-source and available for download at https://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc.
- KLEE [34]: The KLEE LLVM Execution Engine is a symbolic virtual machine for LLVM bytecode, although the main focus are programs written in the C language. KLEE supports a large amount of system calls through a library of models, allowing the software under test to interact with the environment. As one of the most popular open-source symbolic execution tools, many extensions of KLEE are available, such as Cloud9 (distributed symbolic execution, available at http://cloud9.epfl.ch/). KLEE is available for download at https://klee.github.io/.
- JBSE [29]: JBSE is a symbolic virtual machine for JVM bytecodes, in a similar fashion to SPF. The main feature of JBSE is the support for complex, user-friendly assumptions over the heap shape of the program to cull the search space. Users can annotate methods that should be used to validate new objects (*repOK*), specify LICS rules over symbolic references [27] and trigger instrumentation

methods when new clauses containing specific references are added to the path condition. The tool is open-source and available at https://github.com/pietrobraione/jbse.

- KeY [3]: KeY is a program verification platform for Java programs. Among other features, KeY contains a symbolic execution engine and a visual symbolic debugger that allows the user to step through the resulting symbolic tree. The tool is open-source and available at https://www.key-project.org/.
- S2E [43]: S2E is a platform for symbolic execution of large systems, such as the combination of binaries + kernel + drivers. Unlike most other symbolic execution tools, S2E performs *in-vivo* analysis through the use of virtualization and dynamic translation of x86/ARM instructions. S2E ameliorates the path explosion problem by interleaving concrete/symbolic execution at different parts of the stack (*selective symbolic execution*) and allowing users to relax the consistency models of the analysis to improve scalability. The tool is open-source (but not free for commercial use) and available at http://s2e.systems/
- SAGE [26]: SAGE is a dynamic symbolic execution tool for x86 binaries utilized internally at Microsoft. SAGE is likely the largest deployment of symbolic execution for test generation, with 3.4 billion constraints generated and solved as of 2014. A commercial service offered by Microsoft that leverages SAGE, named "Security Risk Detection", is available for preview at https://www.microsoft.com/en-us/security-risk-detection.
- Jalangi2: Jalangi2 is a platform for program analysis of Javascript programs. Developers can use Jalangi2 to instrument the source code of their programs, which can then be executed with any out-of-shelf Javascript interpreter (Node.js or a browser). The source code can be found at https://github.com/Samsung/jalangi2.
- Manticore: Manticore is a symbolic execution tool for x86/ARM binaries and Ethereum smart contracts. It supports both out-of-box usage and can be customized through its python interface. The tool is open source and can be found at https://github.com/trailofbits/manticore/.
- FuzzBALL: FuzzBALL is a symbolic execution tool for x86/ARM binaries. FuzzBALL is built on top of the Vine [163] intermediate language/static analysis library. Source code is available at http://bitblaze.cs.berkeley.edu/fuzzball.html
- Angr [158]: Angr is a python platform for program analysis and reverse engineering of x86 binaries, developed for the DARPA Cyber Grand Challenge competition. Among more traditional features, Angr provides tools for automatic exploit generation and patching of binaries and a management interface. The platform is open source and available at http://angr.io/.
- Triton [152]: Triton is a dynamic symbolic execution tool for x86/ARM binaries. Features include a taint propagation engine and a python API for customization. The tool is open source and can be found at https://triton.quarkslab.com/.
- JDart [114]: JDart is a dynamic symbolic execution tool for Java programs, built on top of Java PathFinder. The tool is designed to allow developers to replace existing components, such as search strategies or the bytecode semantics, by custom implementations in a easy way. Other features (provided by extensions) include generation of abstract method summaries and the creation of JUnit test suites. JDart is open source, and can be downloaded at: https://github.com/psycopaths/jdart
- CIVL [160]: CIVL is a framework for the specification and verification of programs written in CIVL-C, an intermediate language similar to C with additional primitives for concurrency. The framework include tools such as a symbolic execution-based model checker (capable of checking safety properties and functional equivalence between two programs) and translators for other languages/APIs (e.g. OpenMP, CUDA, Pthreads). CIVL is open source and can be downloaded at https://vsl.cis.udel.edu/civl/.

XII. Conclusion

This chapter has reviewed the latest advancements in symbolic execution in the last five years. Specifically, we have presented recent advancements in addressing the main challenges of symbolic execution such as constraint solving, path explosion, concurrency, etc., and we have also reported advancements in three application areas of symbolic execution, i.e., testing, security, and probabilistic program analysis. That said, we have only sketched a few important areas of advancements here, since there is a lot of work related to this subject and it is impossible to include everything in one article. More work can be found in previous surveys [35], [37], [42], [137] for a complete review of symbolic execution, especially for the advancements before five years ago.

References

[1] P. A. Abdulla, M. F. Atig, Y. F. Chen, L. Hol??k, A. Rezine, P. R??mmer, and J. Stenman. String constraints for verification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8559 LNCS:150–166, 2014.

[2] P. A. Abdulla, M. F. Atig, Y. F. Chen, L. Hol??k, A. Rezine, P. R??mmer, and J. Stenman. Norn: An SMT solver for string constraints. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9206, pages 462–469. Springer, Cham, 2015.

[3] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[4] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, page 1, 2012.

[6] A. Aquino, F. A. Bianchi, M. Chen, G. Denaro, M. Pezzè, and V. G. Buffi. Reusing Constraint Proofs in Program Analysis. *International Symposium on Software Testing and Analysis (ISSTA 2015)*, (ii):305–315, 2015.

[7] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 1083–1094, 2014.

[8] A. Aydin, L. Bang, and T. Bultan. Automata-based model counting for string constraints. In D. Kroening and C. S. Păsăreanu, editors, *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 255–272, Cham, 2015. Springer International Publishing.

[9] J. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7976 LNCS:99–116, 2013.

[10] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb. Symbolic path-oriented test data generation for floating-point programs. *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, pages 1–10, 2013.

[11] C. Baier, J.-P. Katoen, and K. G. Larsen. *Principles of model checking*. MIT press, 2008.

[12] V. Baldoni, N. Berline, J. A. De Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, M. Vergne, and J. Wu. A user's guide for latte integrale v1. 7.2. 2014.

[13] K. Bansal, , A. Reynolds, , T. King, , C. Barrett, , and T. Wies. Deciding Local Theory Extensions via E-matching. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 87—-105, 2015.

[14] S. Bardin, N. Kosmatov, and F. Cheynier. Efficient leveraging of symbolic execution to advanced coverage criteria. *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*, 7:173–182, 2014.

[15] A. Barvinok and J. E. Pommersheim. An algorithmic theory of lattice points. *New perspectives in algebraic combinatorics*, 38:91, 1999.

[16] J. Belt, Robby, P. Chalin, J. Hatcliff, and X. Deng. Efficient Symbolic Execution of Value-Based Data Structures for Critical Systems. In *NASA Formal Methods: 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, pages 295–309. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[17] T. Bergan, L. Ceze, and D. Grossman. Input-covering schedules for multithreaded programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 677–692, New York, NY, USA, 2013. ACM.

[18] T. Bergan, D. Grossman, and L. Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 491–506, New York, NY, USA, 2014. ACM.

[19] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[20] N. Bjørner, N. Tillmann, and A. Voronkov. *Path Feasibility Analysis for String-Manipulating Programs*, pages 307–321. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[21] S. Blackshear, B.-Y. E. Chang, and M. Sridharan. Thresher: Precise refutations for heap reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 275–286, New York, NY, USA, 2013. ACM.

[22] M. Borges, M. D'Amorim, S. Anand, D. Bushnell, and C. S. Păsăreanu. Symbolic execution with interval solving and meta-heuristic search. *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, (1):111–120, 2012.

[23] M. Borges, A. Filieri, M. D'Amorim, and C. Păsăreanu. Iterative distribution-aware sampling for probabilistic symbolic execution. *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, pages 866–877, 2015.

[24] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser. Compositional solution space quantification for probabilistic software analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 123–132, New York, NY, USA, 2014. ACM.

[25] M. Borges, Q.-S. Phan, A. Filieri, and C. S. Păsăreanu. Model-counting approaches for nonlinear numerical constraints. In C. Barrett, M. Davies, and T. Kahsai, editors, *NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 131–138, Cham, 2017. Springer International Publishing.

[26] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 122–131, Piscataway, NJ, USA, 2013. IEEE Press.

[27] P. Braione, G. Denaro, and M. Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 411, 2013.

[28] P. Braione, G. Denaro, and M. Pezzè. Symbolic Execution of Programs with Heap Inputs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 602–613, 2015.

[29] P. Braione, G. Denaro, and M. Pezzè. Symbolic Execution of Programs with Heap Inputs. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 602–613, 2015.

[30] P. Braione, G. Denaro, and M. Pezzè. JBSE: a symbolic executor for Java programs with complex heap inputs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 1018–1022, New York, New York, USA, 2016. ACM Press.

[31] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, Sept. 1992.

[32] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM.

[33] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[34] 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*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[35] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, 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, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 1066–1071, 2011.

[36] C. Cadar and H. Palikareva. Shadow Symbolic Execution for Better Testing of Evolving Software. *Icse*, pages 432–435, 2014.

[37] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[38] B. Carre and J. Garnsworthy. Spark - an annotated ada subset for safety-critical programming. pages 392 – 402, Baltimore, MD, United states, 1990.

[39] S. Chakraborty, K. S. Meel, and M. Y. Vardi. *A Scalable Approximate Model Counter*, pages 200–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[40] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li. Symcerts: Practical symbolic execution for exposing noncompliance in x.509 certificate validation implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 503–520, May 2017.

[41] B. Chen, Y. Liu, and W. Le. Generating performance distributions via probabilistic symbolic execution. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 49–60, May 2016.

[42] T. Chen, X.-S. Zhang, S.-Z. Guo, H.-Y. Li, and Y. Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Gener. Comput. Syst.*, 29(7):1758–1773, Sept. 2013.

[43] V. Chipounov, V. Kuznetsov, and G. Candea. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, Feb. 2012.

[44] D. Chistikov, R. Dimitrova, and R. Majumdar. Approximate counting in smt and value estimation for probabilistic programs. *Acta Informatica*, 54(8):729–764, Dec 2017.

[45] M. Christakis, P. Müller, and V. Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, pages 144–155, 2016.

[46] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[47] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, ACM '76, pages 488–491, 1976.

[48] D. Coughlin, B.-Y. E. Chang, A. Diwan, and J. G. Siek. Measuring enforcement windows with symbolic trace interpretation: what well-behaved programs say. *Proc. ISSTA*, (May):276–286, 2012.

[49] M. Cristi{\'a}, , and G. Rossi. A Decision Procedure for Sets, Binary Relations and Partial Functions. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 1, pages 179—-198, 2016.

[50] P. Daca, T. A. Henzinger, and A. Kupriyanov. Array folds logic. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9780, pages 230–248, 2016.

[51] P. Dan Marinescu and C. Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. *Proceedings - International Conference on Software Engineering*, pages 716–726, 2012.

[52] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. {FIE} on Firmware: Finding Vulnerabilities in Embedded Systems using Symbolic Execution Finding Vulnerabilities in Embedded Systems using Symbolic Execution. *Proceedings of the 22nd USENIX Security Symposium*, pages 463–478, 2013.

[53] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 166–177, Washington, DC, USA, 2015. IEEE Computer Society.

[54] M. Dhok, M. K. Ramanathan, and N. Sinha. Type-aware concolic testing of JavaScript programs. *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, pages 168–179, 2016.

[55] M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, Z. Rakamarić, and V. Raman. The Dart, the Psyco, and the Doop. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–5, 2015.

[56] P. Dinges and G. Agha. Solving Complex Path Conditions Through Heuristic Search on Induced Polytopes. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (C):425–436, 2014.

[57] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 151–162, New York, NY, USA, 2007. ACM.

[58] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 37–47, New York, NY, USA, 2013. ACM.

[59] A. Filieri, M. F. Frias, C. S. Păsăreanu, and W. Visser. Model counting for complex data structures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9232:222–241, 2015.

[60] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in Symbolic PathFinder. *Proceedings - International Conference on Software Engineering*, pages 622–631, 2013.

[61] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys. Statistical Symbolic Execution with Informed Sampling. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 437–448, 2014.

[62] A. Fromherz, K. S. Luckow, and C. S. Păsăreanu. Symbolic Arrays in Symbolic PathFinder. *ACM SIGSOFT Software Engineering Notes*, 41(6):1–5, jan 2016.

[63] Z. Fu and Z. Su. Xsat: A fast floating-point satisfiability solver. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9780, pages 187–209, 2016.

[64] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. *Proceedings - International Conference on Software Engineering*, pages 132–141, 2013.

[65] J. Geldenhuys, N. Aguirre, M. F. Frias, and W. Visser. Bounded Lazy Initialization. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods: 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, pages 229–243. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[66] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*, page 166, 2012.

[67] P. Godefroid. Compositional dynamic test generation. *SIGPLAN Not.*, 42(1):47–54, Jan. 2007.

[68] 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*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[69] 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*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[70] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. *SIGPLAN Not.*, 45(1):43–56, Jan. 2010.

[71] C. P. Gomes, A. Sabharwal, and B. Selman. *Model Counting*, pages 633–654. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[72] L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: An interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, Mar. 2006.

[73] S. Guo, M. Kusano, and C. Wang. Conc-ise: Incremental symbolic execution of concurrent software. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 531–542, New York, NY, USA, 2016. ACM.

[74] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 854–865, New York, NY, USA, 2015. ACM.

[75] L. Hadarean, K. Bansal, D. Jovanovi??, C. Barrett, and C. Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8559 LNCS:680–695, 2014.

[76] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowser : a guided fuzzer to find buffer overflow vulnerabilities. *SEC'13 Proceedings of the 22nd USENIX conference on Security*, pages 49–64, 2013.

[77] N. Hasabnis and R. Sekar. Extracting instruction semantics via symbolic execution of code generators. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 301–313, 2016.

[78] V. Herdt, H. M. Le, D. Große, and R. Drechsler. *ParCoSS: Efficient Parallelized Compiled Symbolic Simulation*, pages 177–183. Springer International Publishing, Cham, 2016.

[79] D. Holling, S. Banescu, M. Probst, A. Petrovska, and A. Pretschner. Nequivack: Assessing Mutation Score Confidence. *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2016*, pages 152–161, 2016.

[80] D. Honfi, A. V??r??s, and Z. Micskei. SEViz: A tool for visualizing symbolic execution. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, 2015.

[81] J. Jaffar, V. Murali, and J. a. Navas. Boosting concolic testing via interpolation. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, pages 53–63, 2013.

[82] K. Jamrozik, G. Fraser, N. Tillmann, and J. De Halleux. Augmented dynamic symbolic execution. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, page 254, 2012.

[83] S. Jana, Y. Kang, S. Roth, and B. Ray. Automatically Detecting Error Handling Bugs Using Error Specifications. *USENIX Security*, pages 345–362, 2016.

[84] C. S. Jensen, M. R. Prasad, and A. M?ller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, page 67, New York, New York, USA, 2013. ACM Press.

[85] X. Jia, C. Ghezzi, and S. Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, volume abs/1501.0, pages 177–187, New York, New York, USA, 2015. ACM Press.

[86] Y. Jia and M. Harman. Higher order mutation testing. *Inf. Softw. Technol.*, 51(10):1379–1393, Oct. 2009.

[87] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[88] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[89] K. Kähkönen, O. Saarikivi, and K. Heljanko. Using unfoldings in automated testing of multithreaded programs. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 150–159, New York, NY, USA, 2012. ACM.

[90] S. Kausler and E. Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, number August, pages 259–270, New York, New York, USA, 2014. ACM Press.

[91] S. Kausler and E. Sherman. User-defined backtracking criteria for symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, 2014.

[92] R. Kersten, S. Person, N. Rungta, and O. Tkachuk. Improving Coverage of Test Cases Generated by Symbolic PathFinder for Programs with Loops. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–5, 2015.

[93] S. Khurshid, C. S. Păsăreanu, and W. Visser. *Generalized Symbolic Execution for Model Checking and Testing*, pages 553–568. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[94] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, New York, NY, USA, 2009. ACM.

[95] Y. Kim, M. Kim, Y. J. Kim, and Y. Jang. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. *Proceedings - International Conference on Software Engineering*, pages 1143–1152, 2012.

[96] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[97] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[98] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. *ACM SIGPLAN Notices*, 47(6):193–204, 2012.

[99] Q. L. Le, J. Sun, and W.-N. Chin. Satisfiability Modulo Heap-Based Programs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9779, pages 382–404. 2016.

[100] W. Le. Segmented symbolic analysis. *Proceedings - International Conference on Software Engineering*, pages 212–221, 2013.

[101] W. Le and S. D. Pattison. Patch verification via multiversion interprocedural control flow graphs. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 1047–1058, 2014.

[102] G. Li, E. Andreasen, and I. Ghosh. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 449–459, 2014.

[103] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. Gklee: Concolic verification and test generation for gpus. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 215–224, New York, NY, USA, 2012. ACM.

[104] X. Li, Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li. Symbolic execution of complex program driven by machine learning based constraint solving. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 554–559, 2016.

[105] Y. Li, S. C. Cheung, X. Zhang, and Y. Liu. Scaling Up Symbolic Analysis by Removing Z-Equivalent States. *Tsem14*, 23(4):1–32, 2014.

[106] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. *ACM SIGPLAN Notices*, 48(10):19–32, 2013.

[107] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8559 LNCS:646–662, 2014.

[108] Y. Lin and T. Miller. Looking closer at compositional symbolic execution. In *Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference*, ASWEC ' 15 Vol. II, pages 138–140, New York, NY, USA, 2015. ACM.

[109] Y. Lin, T. Miller, and H. Sondergaard. Compositional symbolic execution using fine-grained summaries. In *Proceedings of the 2015 24th Australasian Software Engineering Conference (ASWEC)*, ASWEC '15, pages 213–222, Washington, DC, USA, 2015. IEEE Computer Society.

[110] P. Liu, O. Tripp, and X. Zhang. Ipa: Improving predictive analysis with pointer analysis. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 59–69, New York, NY, USA, 2016. ACM.

[111] J. Lloyd and E. Sherman. Minimizing the Size of Path Conditions Using Convex Polyhedra Abstract Domain. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–5, 2015.

[112] B. Loring, D. Mitchell, and J. Kinder. ExpoSE : Practical Symbolic Execution of Standalone JavaScript. (i), 2017.

[113] K. Luckow, M. Dimjašević, and D. Giannakopoulou. JDart : A Dynamic Symbolic Analysis Framework.

[114] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman. JDart: A dynamic symbolic analysis framework. In M. Chechik and J.-F. Raskin, editors, *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9636 of *Lecture Notes in Computer Science*, pages 442–459. Springer, 2016.

[115] K. Luckow, C. S. Păsăreanu, M. B. Dwyer, A. Filieri, and W. Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pages 575–586, 2014.

[116] K. S. Luckow and C. S. Păsăreanu. Symbolic PathFinder v7. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, 2014.

[117] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 565–576, New York, NY, USA, 2014. ACM.

[118] M. S. Mahmood, M. Abdul Ghafoor, and J. H. Siddiqui. Symbolic execution of stored procedures in database management systems. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 519–530, 2016.

[119] S. Makhdoom, M. A. Khan, and J. H. Siddiqui. Incremental symbolic execution for automated test suite maintenance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pages 271–276, New York, New York, USA, 2014. ACM Press.

[120] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings of the Fourth International Workshop on Computer Aided Verification*, CAV '92, pages 164–177, London, UK, UK, 1993. Springer-Verlag.

[121] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. SIG-Droid: Automated system input generation for Android applications. *2015 IEEE 26th International Symposium on Software Reliability Engineering, ISSRE 2015*, pages 461–471, 2016.

[122] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1, 2012.

[123] S. Monpratarnchai, S. Fujiwara, A. Katayama, and T. Uehara. Automated testing for Java programs using JPF-based test case generation. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, 2014.

[124] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9779, pages 405–425. 2016.

[125] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[126] C. Nguyen, H. Yoshida, M. Prasad, I. Ghosh, and K. Sen. Generating succinct test cases using don't care analysis. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, 2015.

[127] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen. Dangling references in multi-configuration and dynamic PHP-based Web applications. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pages 399–409, 2013.

[128] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer. Macke: Compositional analysis of low-level vulnerabilities with symbolic execution. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 780–785, Sept 2016.

[129] A. Orso and G. Rothermel. Software testing: A research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 117–132. ACM, 2014.

[130] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[131] C. S. Păsăreanu, Q. S. Phan, and P. Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 387–400, June 2016.

[132] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*, page 504, 2011.

[133] Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. D'Amorim. Quantifying information leaks using reliability analysis. *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software - SPIN 2014*, pages 105–108, 2014.

[134] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8044 LNCS:773–789, 2013.

[135] M. L. Potet, L. Mounier, M. Puys, and L. Dureuil. Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*, pages 213–222, 2014.

[136] I. Pustogarov, T. Ristenpart, and V. Shmatikov. Using program analysis to synthesize sensor spoofing attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 757–770, New York, NY, USA, 2017. ACM.

[137] 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.*, 11(4):339–353, Oct. 2009.

[138] D. Qi, H. D. Nguyen, and A. Roychoudhury. Path exploration based on symbolic output. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering SE - ESEC/FSE '11*, 22(4):278–288, 2011.

[139] R. Qiu. *Scaling and certifying symbolic execution*. PhD dissertation, University of Texas at Austin, 2016.

[140] R. Qiu, S. Khurshid, C. S. Pasareanu, and G. Yang. A synergistic approach for distributed symbolic execution using test ranges. In *ICSE '17 - Companion*, pages 130–132, 2017.

[141] R. Qiu, G. Yang, C. S. Păsăreanu, and S. Khurshid. Compositional symbolic execution with memoized replay. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 632–642, Piscataway, NJ, USA, 2015. IEEE Press.

[142] D. A. Ramos and D. Engler. Under-Constrained Symbolic Execution : Correctness Checking for Real Code. *USENIX Security Symposium*, pages 49–64, 2015.

[143] E. J. Rapos and J. Dingel. Using Fuzzy Logic and Symbolic Execution to Prioritize UML-RT Test Cases. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[144] N. Razavi, F. Ivancic, V. Kahlon, and A. Gupta. Concurrent test generation using concolic multi-trace analysis. In *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, pages 239–255, 2012.

[145] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[146] J. M. Rojas and C. S. Pasareanu. Compositional symbolic execution through program specialization. 2013.

[147] A. Romano and D. Engler. Expression reduction from programs in a symbolic binary executor. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7976 LNCS:301–319, 2013.

[148] A. Romano and D. R. Engler. symMMU: Symbolically Executed Runtime Libraries for Symbolic Memory Access. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pages 247–258, New York, New York, USA, 2014. ACM Press.

[149] N. Rosner, J. Geldenhuys, N. Aguirre, W. Visser, and M. Frias. BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support. *IEEE Transactions on Software Engineering*, 41(7):1–1, 2015.

[150] G. Rubino and B. Tuffin. *Rare event simulation using Monte Carlo methods*. John Wiley & Sons, 2009.

[151] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an smt-based analysis. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 313–327, Berlin, Heidelberg, 2011. Springer-Verlag.

[152] F. Saudel and J. Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.

[153] D. Scheurer and H. Reiner. A General Lattice Model for Merging Symbolic Execution Branches. 10009:57–73, 2016.

[154] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[155] K. Sen, G. Necula, L. Gong, and W. Choi. MultiSE: multi-path symbolic execution using value summaries. *Joint Meeting on Foundations of Software Engineering*, pages 842–853, 2015.

[156] H. Seo and S. Kim. How We Get There: A Context-guided Search Strategy in Concolic Testing. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 413–424, 2014.

[157] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[158] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[159] J. H. Siddiqui and S. Khurshid. Scaling Symbolic Execution Using Ranged Analysis. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 523–536, 2012.

[160] S. F. Siegel, M. B. Dwyer, G. Gopalakrishnan, Z. Luo, Z. Rakamaric, R. Thakur, M. Zheng, and T. K. Zirkel. Civl: The concurrency intermediate verification language. Technical Report UD-CIS-2014/001, Department of Computer and Information Sciences, University of Delaware, 2014.

[161] S. Siroky, R. Podorozhny, and G. Yang. Verification of Architectural Constraints on Sequences of Method Invocations. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–4, 2015.

[162] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 387–400, New York, NY, USA, 2012. ACM.

[163] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. *BitBlaze: A New Approach to Computer Security via Binary Analysis*, pages 1–25. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[164] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Ndss*, number February, pages 21–24, 2016.

[165] T. Stoenescu, A. S. B, S. Predut, and F. Ipate. RIVER: A Binary Analysis Framework Using Symbolic Execution and Reversible x86 Instructions. volume 9995, pages 779–785, 2016.

[166] J. Strejček and M. Trtík. Abstracting path conditions. *Proc. ISSTA*, pages 155–165, 2012.

[167] T. Su, Z. Fu, G. Pu, J. He, and Z. Su. Combining symbolic execution and model checking for data flow testing. *Proceedings - International Conference on Software Engineering*, 1:654–665, 2015.

[168] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[169] A. Tiwari and P. Lincoln. A Nonlinear Real Arithmetic Fragment. pages 729–736, 2014.

[170] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, New York, NY, USA, 2014. ACM.

[171] M. T. Trinh, D. H. Chu, and J. Jaffar. Progressive reasoning over recursively-defined strings. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9779, pages 218–240, 2016.

[172] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, Mar. 2007.

[173] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, page 1, New York, New York, USA, 2012. ACM Press.

[174] K. Vorobyov and P. Krishnan. Combining static analysis and constraint solving for automatic test case generation. *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, pages 915–920, 2012.

[175] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, pages 256–272, Berlin, Heidelberg, 2009. Springer-Verlag.

[176] R. Wang, P. Ning, T. Xie, and Q. Chen. MetaSymploit: Day-One Defense against Script-based Attacks with Security-Enhanced Symbolic Analysis. *Proceedings of the USENIX Security Symposium*, 2013.

[177] M. Whalen, G. Gay, D. You, M. P. E. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 102–111, May 2013.

[178] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan. DASE: Document-assisted symbolic execution for improving automated software testing. *Proceedings - International Conference on Software Engineering*, 1:620–631, 2015.

[179] X. Xie, Y. Liu, W. Le, X. Li, and H. Chen. S-looper: automatic summarization for multipath string loops. *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, pages 188–198, 2015.

[180] D. Xu, J. Ming, and D. Wu. Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 921–937, 2017.

[181] G. Yang, S. Khurshid, S. Person, and N. Rungta. Property differencing for incremental checking. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 1059–1070, 2014.

[182] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*, page 144, New York, New York, USA, 2012. ACM Press.

[183] G. Yang, S. Person, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. *ACM Transactions on Software Engineering and Methodology*, 24(1):3:1–3:42, 2014.

[184] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao. Postconditioned symbolic execution. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, 2015.

[185] H. Yoshida, S. Tokumoto, M. R. Prasad, I. Ghosh, and T. Uehara. FSX: fine-grained incremental unit test generation for C/C++ programs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, pages 106–117, New York, New York, USA, 2016. ACM Press.

[186] D. You, S. Rayadurgam, M. Whalen, M. P. E. Heimdahl, and G. Gay. Efficient observability-based test generation by dynamic symbolic execution. *2015 IEEE 26th International Symposium on Software Reliability Engineering, ISSRE 2015*, pages 228–238, 2016.

[187] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik. APISan: Sanitizing API Usages through Semantic Cross-Checking. *USENIX Security Symposium*, pages 363–378, 2016.

[188] P. Zhang, S. Elbaum, and M. B. Dwyer. Compositional load test generation for software pipelines. *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*, page 89, 2012.

[189] Y. Zhang, Z. Chen, and J. Wang. S2PF Speculative Symbolic PathFinder. *ACM SIGSOFT Software Engineering Notes*, 37(6):1, 2012.

[190] Y. Zhang, Z. Chen, and J. Wang. Speculative symbolic execution. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pages 101–110, 2012.

[191] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu. Regular property guided dynamic symbolic execution. *Proceedings - International Conference on Software Engineering*, 1:643–653, 2015.

[192] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 114–124, 2013.

[193] H. Zhu, G. Petri, and S. Jagannathan. Poling: SMT aided linearizability proofs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9207, pages 3–19, 2015.