A Survey of Symbolic Execution Techniques

ROBERTO BALDONI, Cyber Intelligence and Information Security Research Center, Sapienza
EMILIO COPPA, SEASON Lab, Sapienza University of Rome
DANIELE CONO D'ELIA, SEASON Lab, Sapienza University of Rome
CAMIL DEMETRESCU, SEASON Lab, Sapienza University of Rome
IRENE FINOCCHI, SEASON Lab, Sapienza University of Rome

Many security and software testing applications require checking whether certain properties of a program hold for any possible usage scenario. For instance, a tool for identifying software vulnerabilities may need to rule out the existence of any backdoor to bypass a program's authentication. One approach would be to test the program using different, possibly random inputs. As the backdoor may only be hit for very specific program workloads, automated exploration of the space of possible inputs is of the essence. Symbolic execution provides an elegant solution to the problem, by systematically exploring many possible execution paths at the same time without necessarily requiring concrete inputs. Rather than taking on fully specified input values, the technique abstractly represents them as symbols, resorting to constraint solvers to construct actual instances that would cause property violations. Symbolic execution has been incubated in dozens of tools developed over the last four decades, leading to major practical breakthroughs in a number of prominent software reliability applications. The goal of this survey is to provide an overview of the main ideas, challenges, and solutions developed in the area, distilling them for a broad audience.

CCS Concepts: •**Software and its engineering** → **Software verification;** *Software testing and debugging;* •**Security and privacy** → Software and application security;

Additional Key Words and Phrases: Symbolic execution, static analysis, concolic execution, software testing

ACM Reference Format:
Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi, 2016. A survey of symbolic execution techniques. *ACM Comput. Surv.* 0, 0, Article 0 (0000), 37 pages.
DOI: 0000001.0000001

"Sometimes you can't see how important something is in its moment, even if it seems kind of important. This is probably one of those times."

(Cyber Grand Challenge highlights from DEF CON 24, August 6, 2016)

1. INTRODUCTION

Symbolic execution is a popular program analysis technique introduced in the mid '70s to test whether certain properties can be violated by a piece of software [King 1975; Boyer et al. 1975; King 1976; Howden 1977]. Aspects of interest could be that no division by zero is ever performed, no `NULL` pointer is ever dereferenced, no backdoor exists that can bypass authentication, etc. While in general there is no automated way to decide some properties (e.g., the target of an indirect jump), heuristics and approximate analyses can prove useful in practice in a variety of settings, including mission-critical and security applications.

Author's addresses: R. Baldoni, E. Coppa, D.C. D'Elia, and C. Demetrescu, Department of Computer, Control, and Management Engineering, Sapienza University of Rome; I. Finocchi, Department of Computer Science, Sapienza University of Rome. This work is supported in part by a grant of the Italian Presidency of the Council of Ministers and by the CINI National Laboratory of Cyber Security.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 0000 ACM. 0360-0300/0000/-ART0 $15.00
DOI: 0000001.0000001

```
1.  void foobar(int a, int b) {
2.     int x = 1, y = 0;
3.     if (a != 0) {
4.        y = 3+x;
5.        if (b == 0)
6.           x = 2*(a+b);
7.     }
8.     assert(x-y != 0);
9.  }
```

Fig. 1: Warm-up example: which values of a and b make the assert fail?

In a concrete execution, a program is run on a specific input and a single control flow path is explored. Hence, in most cases concrete executions can only under-approximate the analysis of the property of interest. In contrast, symbolic execution can simultaneously explore multiple paths that a program could take under different inputs. This paves the road to sound analyses that can yield strong guarantees on the checked property. The key idea is to allow a program to take on *symbolic* – rather than concrete – input values. Execution is performed by a *symbolic execution engine*, which maintains for each explored control flow path: (i) a first-order Boolean *formula* that describes the conditions satisfied by the branches taken along that path, and (ii) a *symbolic memory store* that maps variables to symbolic expressions or values. Branch execution updates the formula, while assignments update the symbolic store. A *model checker*, typically based on a *satisfiability modulo theories* (SMT) solver [Barrett et al. 2014], is eventually used to verify whether there are any violations of the property along each explored path and if the path itself is realizable, i.e., if its formula can be satisfied by some assignment of concrete values to the program's symbolic arguments.

Symbolic execution techniques have been brought to the attention of a heterogeneous audience since DARPA announced in 2013 the Cyber Grand Challenge, a two-year competition seeking to create automatic systems for vulnerability detection, exploitation, and patching in near real-time [Shoshitaishvili et al. 2016].

More remarkably, symbolic execution tools have been running 24/7 in the testing process of many Microsoft applications since 2008, revealing for instance nearly 30% of all the bugs discovered by file fuzzing during the development of Windows 7, which other program analyses and blackbox testing techniques missed [Godefroid et al. 2012].

In this article, we survey the main aspects of symbolic execution and discuss the most prominent techniques employed for instance in software testing and computer security applications. Our discussion is mainly focused on *forward* symbolic execution, where a symbolic engine analyzes many paths simultaneously starting its exploration from the main entry point of a program. We start with a simple example that highlights many of the fundamental issues addressed in the remainder of the article.

1.1. A Warm-Up Example

Consider the C code of Figure 1 and assume that our goal is to determine which inputs make the assert at line 8 of function foobar fail. Since each 4-byte input parameter can take as many as 2^{32} distinct integer values, the approach of running concretely function foobar on randomly generated inputs will unlikely pick up exactly the assert-failing inputs. By evaluating the code using symbols for its inputs, instead of concrete values, symbolic execution overcomes this limitation and makes it possible to reason on *classes of inputs*, rather than single input values.

In more detail, every value that cannot be determined by a static analysis of the code, such as an actual parameter of a function or the result of a system call that reads data from a stream, is represented by a symbol α_i. At any time, the symbolic execution engine maintains a state $(stmt,\ \sigma,\ \pi)$ where:

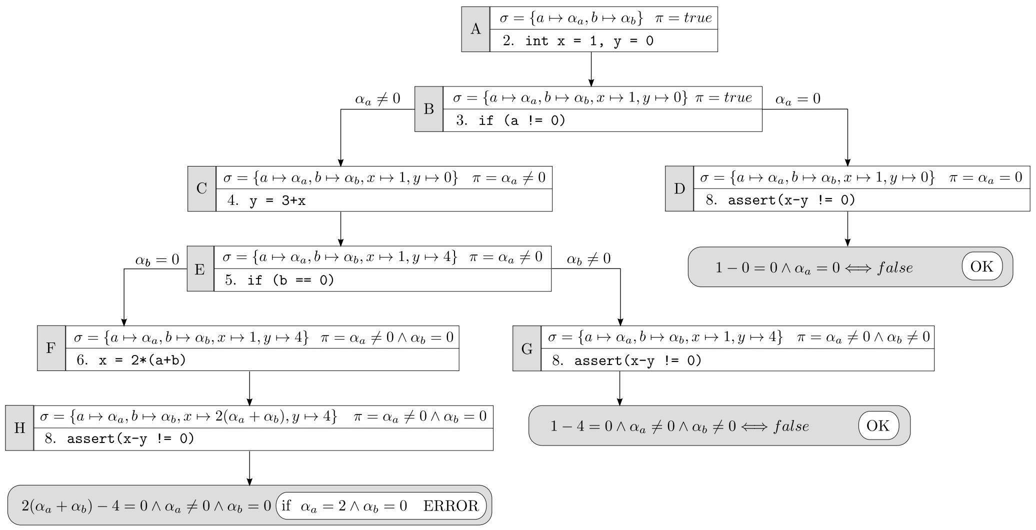

A
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\}$ $\pi = true$
2. int x = 1, y = 0
B
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = true$
3. if (a != 0)
$\alpha_a \neq 0$
$\alpha_a = 0$
C
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a \neq 0$
4. y = 3+x
D
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a = 0$
8. assert(x-y != 0)
$1 - 0 = 0 \wedge \alpha_a = 0 \Longleftrightarrow false$
OK
E
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0$
5. if (b == 0)
$\alpha_b = 0$
$\alpha_b \neq 0$
F
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b = 0$
6. x = 2*(a+b)
G
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b \neq 0$
8. assert(x-y != 0)
$1 - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b \neq 0 \Longleftrightarrow false$
OK
H
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2(\alpha_a + \alpha_b), y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b = 0$
8. assert(x-y != 0)
$2(\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0$
if $\alpha_a = 2 \wedge \alpha_b = 0$ ERROR

Fig. 2: Symbolic execution tree of function `foobar` given in Figure 1. Each execution state, labeled with an upper case letter, shows the statement to be executed, the symbolic store σ, and the path constraints π. Leaves are evaluated against the condition in the `assert` statement.

— $stmt$ is the next statement to evaluate. For the time being, we assume that $stmt$ can be an assignment, a conditional branch, or a jump (more complex constructs such as function calls and loops will be discussed in Section 5).
— σ is a *symbolic store* that associates program variables with either expressions over concrete values or symbolic values α_i.
— π denotes the *path constraints*, i.e., is a formula that expresses a set of assumptions on the symbols α_i due to branches taken in the execution to reach $stmt$. At the beginning of the analysis, $\pi = true$.

Depending on $stmt$, the symbolic engine changes the state as follows:
— The evaluation of an assignment $x = e$ updates the symbolic store σ by associating x with a new symbolic expression e_s. We denote this association with $x \mapsto e_s$, where e_s is obtained by evaluating e in the context of the current execution state and can be any expression involving unary or binary operators over symbols and concrete values.
— The evaluation of a conditional branch `if` e `then` s_{true} `else` s_{false} affects the path constraints π. The symbolic execution is forked by creating two execution states with path constraints π_{true} and π_{false}, respectively, which correspond to the two branches: $\pi_{true} = \pi \wedge e_s$ and $\pi_{false} = \pi \wedge \neg e_s$, where e_s is a symbolic expression obtained by evaluating e. Symbolic execution independently proceeds on both states.
— The evaluation of a jump `goto` s updates the execution state by advancing the symbolic execution to statement s.

A symbolic execution of function `foobar`, which can be effectively represented as a tree, is shown in Figure 2. Initially (execution state A) the path constraints are `true` and input arguments `a` and `b` are associated with symbolic values. After initializing local variables `x` and `y` at line 2, the symbolic store is updated by associating `x` and `y` with concrete values 1 and 0, respectively (execution state B). Line 3 contains a conditional branch and the execution is forked: depending on the branch taken, a different statement is evaluated next and different assumptions are made on symbol α_a (execution states C and D, respectively). In the branch where $\alpha_a \neq 0$, variable `y` is assigned with `x` $+ 3$, obtaining $y \mapsto 4$ in state E because $x \mapsto 1$ in state C. In general, arithmetic expression evaluation simply manipulates the symbolic values. After expanding every execution state until the `assert` at line 8 is reached on all branches, we can check which input values for parameters `a` and `b` can make the `assert` fail. By analyzing execution states $\{D, G, H\}$, we can conclude that only H can make `x-y = 0` true. The path constraints for H at this point implicitly define the set of inputs that are unsafe for `foobar`. In particular, any input values such that:

$$2(\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0$$

will make `assert` fail. An instance of unsafe input parameters can be eventually determined by invoking an *SMT solver* [Barrett et al. 2014] to solve the path constraints, which in this example would yield $a = 2$ and $b = 0$.

1.2. Challenges in Symbolic Execution

In the example discussed in Section 1.1 symbolic execution can identify *all* the possible unsafe inputs that make the `assert` fail. This is achieved through an exhaustive exploration of the possible execution states. From a theoretical perspective, exhaustive symbolic execution provides a *sound* and *complete* methodology for any decidable analysis. Soundness prevents false negatives, i.e., all possible unsafe inputs are guaranteed to be found, while completeness prevents false positives, i.e., input values deemed unsafe are actually unsafe. As we will discuss later on, exhaustive symbolic execution is unlikely to scale beyond small applications. Hence, in practice we often settle for less ambitious goals, e.g., by trading soundness for performance.

Challenges that symbolic execution has to face when processing real-world code can be significantly more complex than those illustrated in our warm-up example. Several observations and questions naturally arise:

— *Memory*: how does the symbolic engine handle pointers, arrays, or other complex objects? Code manipulating pointers and data structures may give rise not only to symbolic stored data, but also to addresses being described by symbolic expressions.

— *Environment*: how does the engine handle interactions across the software stack? Calls to library and system code can cause side-effects, e.g., the creation of a file or a call back to user code, that could later affect the execution and must be accounted for. However, evaluating any possible interaction outcome may be unfeasible.

— *State space explosion*: how does symbolic execution deal with path explosion? Language constructs such as loops might exponentially increase the number of execution states. It is thus unlikely that a symbolic execution engine can exhaustively explore all the possible states within a reasonable amount of time.

— *Constraint solving*: what can a constraint solver do in practice? SMT solvers can scale to complex combinations of constraints over hundreds of variables. However, constructs such as non-linear arithmetic pose a major obstacle to efficiency.

Depending on the specific context in which symbolic execution is used, different choices and assumptions are made to address the questions highlighted above. Although these choices typically affect soundness or completeness, in several scenarios a partial exploration of the space of possible execution states may be sufficient to achieve the goal (e.g., identifying a crashing input for an application) within a limited time budget.

1.3. Related Work

Symbolic execution has been the focus of a vast body of literature. As of August 2017, Google Scholar reports 742 articles that include the exact phrase "symbolic execution"

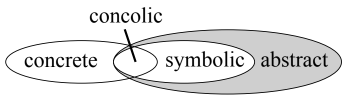


Fig. 3: Concrete and abstract execution machine models.

in the title. Prior to this survey, other authors have contributed technical overviews of the field, such as [Pasareanu and Visser 2009] and [Cadar and Sen 2013]. [Chen et al. 2013] focuses on the more specific setting of automated test generation: it provides a comprehensive view of the literature, covering in depth a variety of techniques and complementing the technical discussions with a number of running examples.

1.4. Organization of the Article

The remainder of this article is organized as follows. In Section 2 we discuss the overall principles and evaluation strategies of a symbolic execution engine. Section 3 through Section 6 address the key challenges that we listed in Section 1.2, while Section 7 discusses how recent advances in other areas could be applied to enhance symbolic execution techniques. Concluding remarks are addressed in Section 8.

2. SYMBOLIC EXECUTION ENGINES

In this section we describe some important principles for the design of symbolic executors and crucial tradeoffs that arise in their implementation. Moving from the concepts of concrete and symbolic runs, we also introduce the idea of *concolic* execution.

2.1. Mixing Symbolic and Concrete Execution

As shown in the warm-up example (Section 1.1), a symbolic execution of a program can generate – in theory – all possible control flow paths that the program could take during its concrete executions on specific inputs. While modeling all possible runs allows for very interesting analyses, it is typically unfeasible in practice, especially on real-world software. A main limitation of classical symbolic execution is that it cannot explore feasible executions that would result in path constraints that cannot be dealt with [Cadar and Sen 2013]. Loss of soundness originates from external code not traceable by the executor, as well as from complex constraints involving, e.g., non-linear arithmetic or transcendental functions. As the time spent in constraint solving is a major performance barrier for an engine, solvability can be intended in the absolute sense, but as in efficiency too. Also, practical programs are typically not self-contained: implementing a symbolic engine able to statically analyze the whole software stack can be rather challenging given the difficulty in accurately evaluating any possible side effect during execution. A fundamental idea to cope with these issues and to make symbolic execution feasible in practice is to mix concrete and symbolic execution: this is dubbed *concolic execution*, where the term concolic is a portmanteau of the words "concrete" and "symbolic" (Figure 3). This general principle has been explored along different angles, discussed in the remainder of this section.

Dynamic Symbolic Execution. One popular concolic execution approach, known as *dynamic symbolic execution* (DSE) or *dynamic test generation* [Godefroid et al. 2005], is to have concrete execution drive symbolic execution. This technique can be very effective in mitigating the issues above. In addition to the symbolic store and the path constraints, the execution engine maintains a concrete store σ_c. After choosing an arbitrary input to begin with, it executes the program both concretely and symbolically by simultaneously updating the two stores and the path constraints. Whenever the concrete execution takes a branch, the symbolic execution is directed toward the same branch and the constraints extracted from the branch condition are added to the current set of path constraints. In short, the symbolic execution is driven by a specific concrete execution. As a consequence, the symbolic engine does not need to invoke the constraint solver to decide whether a branch condition is (un)satisfiable: this is directly tested by the concrete execution. In order to explore different paths, the path conditions given by one or more branches can be negated and the SMT solver invoked to find a satisfying assignment for the new constraints, i.e., to generate a new input. This strategy can be repeated as much as needed to achieve the desired coverage.

Example. Consider the C function in Figure 1 and suppose to choose $a = 1$ and $b = 1$ as input parameters. Under these conditions, the concrete execution takes path $A \rightsquigarrow B \rightsquigarrow C \rightsquigarrow E \rightsquigarrow G$ in the symbolic tree of Figure 2. Besides the symbolic stores shown in Figure 2, the concrete stores maintained in the traversed states are the following:

– $\sigma_c = \{a \mapsto 1,\ b \mapsto 1\}$ in state A;
– $\sigma_c = \{a \mapsto 1,\ b \mapsto 1,\ x \mapsto 1,\ y \mapsto 0\}$ in states B and C;
– $\sigma_c = \{a \mapsto 1,\ b \mapsto 1,\ x \mapsto 1,\ y \mapsto 4\}$ in states E and G.

After checking that the `assert` conditions at line 8 succeed, we can generate a new control flow path by negating the last path constraint, i.e., $\alpha_b \neq 0$. The solver at this point would generate a new input that satisfies the constraints $\alpha_a \neq 0 \wedge \alpha_b = 0$ (for instance $a = 1$ and $b = 0$) and the execution would continue in a similar way along the path $A \rightsquigarrow B \rightsquigarrow C \rightsquigarrow E \rightsquigarrow F$.

Although DSE uses concrete inputs to drive the symbolic execution toward a specific path, it still needs to pick a branch to negate whenever a new path has to be explored. Notice also that each concrete execution may add new branches that will have to be visited. Since the set of non-taken branches across all performed concrete executions can be very large, adopting effective search heuristics (Section 2.3) can play a crucial role. For instance, DART [Godefroid et al. 2005] chooses the next branch to negate using a depth-first strategy. Additional strategies for picking the next branch to negate have been presented in literature. For instance, the *generational search* of SAGE [Godefroid et al. 2008] systematically yet partially explores the state space, maximizing the number of new tests generated while also avoiding redundancies in the search. This is achieved by negating constraints following a specific order and by limiting the backtracking of the search algorithm. Since the state space is only partially explored, the initial input plays a crucial role in the effectiveness of the overall approach. The importance of the first input is similar to what happens in traditional *black-box fuzzing*; hence, symbolic engines such as SAGE are often referred to as *white-box fuzzers*.

The symbolic information maintained during a concrete run can be exploited by the engine to obtain new inputs and explore new paths. The next example shows how DSE can handle invocations to external code that is not symbolically tracked by the concolic engine. Use of concrete values to aid constraint solving will be discussed in Section 6.

Example. Consider function `foo` in Figure 4a and suppose that `bar` is not symbolically tracked by the concolic engine (e.g., it could be provided by a third-party component, written in a different language, or analyzed following a black-box approach). Assuming that $x = 1$ and $y = 2$ are randomly chosen as the initial input parameters, the concolic engine executes `bar` (which returns $a = 0$) and skips the branch that would trigger the error statement. At the same time, the symbolic execution tracks the path constraint $\alpha_y \geq 0$ inside function `foo`. Notice that branch conditions in function `bar` are not known to the engine. To explore the alternative path, the engine negates the path constraint of the branch in `foo`, generating inputs, such as $x = 1$ and $y = -4$, that actually drive the concrete execution to the alternative path. With this approach, the engine can explore both paths in `foo` even if `bar` is not symbolically tracked.

```
void foo(int x, int y) {
   int a = bar(x);
   if (y < 0) ERROR;
}
```
(a)

```
void qux(int x) {
   int a = bar(x);
   if (a > 0) ERROR;
}
```
(b)

```
void baz(int x) {
   abs(&x);
   if (x < 0) ERROR;
}
```
(c)

Fig. 4: Concolic execution: (a) testing of function `foo` even when `bar` cannot be symbolically tracked by an engine, (b) example of false negative, and (c) example of a path divergence, where `abs` drops the sign of the integer at `&x`.

A variant of the previous code is shown in Figure 4b, where function `qux` – differently from `foo` – takes a single input parameter but checks the result of `bar` in the branch condition. Although the engine can track the path constraint in the branch condition tested inside `qux`, there is no guarantee that an input able to drive the execution toward the alternative path is generated: the relationship between a and x is not known to the concolic engine, as `bar` is not symbolically tracked. In this case, the engine could re-run the code using a different random input, but in the end it could fail to explore one interesting path in `qux`.

A related issue is presented by Figure 4c. We observe a *path divergence* when inputs generated for a predicted path lead execution to a different path. In general, this can be due to symbol propagation not being tracked, resulting in inaccurate path constraints, or to imprecision in modeling certain (e.g., bitwise, floating-point) operations in the engine. In the example, function `baz` invokes the external function `abs`, which simply computes the absolute value of a number. Choosing $x = 1$ as the initial concrete value, the concrete execution does not trigger the error statement, but the concolic engine tracks the path constraint $\alpha_x \geq 0$ due to the branch in `baz`, trying to generate a new input by negating it. However the new input, e.g., $x = -1$, does not trigger the error statement due to the (untracked) side effects of `abs`. Interestingly, the engine has no way of detecting that no input can actually trigger the error.

As shown by the example, false negatives (i.e., missed paths) and path divergences are notable downsides of dynamic symbolic execution. DSE trades soundness for performance and implementation effort: false negatives are possible, because some program executions – and therefore possible erroneous behaviors – may be missed, leading to a *complete*, but *under-approximate* form of program analysis. Path divergences are a frequently observed phenomenon: for instance, [Godefroid et al. 2008] reports rates over 60%. [Chen et al. 2015] presents an empirical study of path divergences, analyzing the main patterns that contribute to this phenomenon. External calls, exceptions, type casts, and symbolic pointers are pinpointed as critical aspects of concolic execution that must be carefully handled by an engine to reduce the number of path divergences.

Selective Symbolic Execution. S^{2}E [Chipounov et al. 2012] takes a different approach to mix symbolic and concrete execution based on the observation that one might want to explore only some components of a software stack in full, not caring about others. *Selective symbolic execution* carefully interleaves concrete and symbolic execution, while keeping the overall exploration meaningful.

Suppose a function A calls a function B and the execution mode changes at the call site. Two scenarios arise: (1) *From concrete to symbolic and back*: the arguments of B are made symbolic and B is explored symbolically in full. B is also executed concretely and its concrete result is returned to A. After that, A resumes concretely. (2) *From symbolic to concrete and back*: the arguments of B are concretized, B is executed concretely, and execution resumes symbolically in A. This may impact both soundness and completeness of the analysis: (i) *Completeness*: to make sure that symbolic execution skips any paths that would not be realizable due to the performed concretization (possibly leading to false positives), S^{2}E collects path constraints that keep track of how arguments are concretized, what side effects are made by B, and what return value it produces. (ii) *Soundness*: concretization may cause missed branches after A is resumed (possibly leading to false negatives). To remedy this, the collected constraints are marked as *soft*: whenever a branch after returning to A is made inoperative by a soft constraint, the execution backtracks and a different choice of arguments for B is attempted. To guide re-concretization of B's arguments, S^2E also collects the branch conditions during the concrete execution of B, and chooses the concrete values so that they enable a different concrete execution path in B.

2.2. Design Principles of Symbolic Executors

A number of performance-related design principles that a symbolic execution engine should follow are summarized in [Cha et al. 2012]. Most notably:

(1) *Progress*: the executor should be able to proceed for an arbitrarily long time without exceeding the given resources. Memory consumption can be especially critical, due to the potentially gargantuan number of distinct control flow paths.
(2) *Work repetition*: no execution work should be repeated, avoiding to restart a program several times from its very beginning in order to analyze different paths that might have a common prefix.
(3) *Analysis reuse*: analysis results from previous runs should be reused as much as possible. In particular, costly invocations to the SMT solver on previously solved path constraints should be avoided.

Due to the large size of the execution state space to be analyzed, different symbolic engines have explored different trade-offs between, e.g., running time and memory consumption, or performance and soundness/completeness of the analysis.

Symbolic executors that attempt to execute multiple paths simultaneously in a single run – also called *online* – clone the execution state at each input-dependent branch. Examples are given in KLEE [Cadar et al. 2008], AEG [Avgerinos et al. 2011], S^2E [Chipounov et al. 2012]. These engines never re-execute previous instructions, thus avoiding work repetition. However, many active states need to be kept in memory and memory consumption can be large, possibly hindering progress. Effective techniques for reducing the memory footprint include *copy-on-write*, which tries to share as much as possible between different states [Cadar et al. 2008]. As another issue, executing multiple paths in parallel requires to ensure isolation between execution states, e.g., keeping different states of the OS by emulating the effects of system calls.

Reasoning about a single path at a time, as in concolic execution, is the approach taken by so-called *offline executors*, such as SAGE [Godefroid et al. 2008]. Running each path independently of the others results in low memory consumption with respect to online executors and in the capability of reusing immediately analysis results from previous runs. On the other side, work can be largely repeated, since each run usually restarts the execution of the program from the very beginning. In a typical implementation of offline executors, runs are concrete and require an input seed: the program is first executed concretely, a trace of instructions is recorded, and the recorded trace is then executed symbolically. *Hybrid executors* such as MAYHEM [Cha et al. 2012] attempt at balancing between speed and memory requirements: they start in online mode and generate checkpoints, rather than forking new executors, when memory usage or the number of concurrently active states reaches a threshold. Checkpoints maintain the symbolic execution state and replay information. When a checkpoint is picked for restoration, online exploration is resumed from a restored concrete state.

2.3. Path Selection

Since enumerating all paths of a program can be prohibitively expensive, in many software engineering activities related to testing and debugging the search is prioritized by looking at the most promising paths first. Among several strategies for selecting the next path to be explored, we now briefly overview some of the most effective ones.

We remark that path selection heuristics are often tailored to help the symbolic engine achieve specific goals (e.g., overflow detection). Finding a universally optimal strategy remains an open problem.

Depth-first search (DFS), which expands a path as much as possible before backtracking to the deepest unexplored branch, and *breadth-first search* (BFS), which expands all paths in parallel, are the most common strategies. DFS is often adopted when memory usage is at a premium, but is hampered by paths containing loops and recursive calls. Hence, in spite of the higher memory pressure and of the long time required to complete the exploration of specific paths, some tools resort to BFS, which allows the engine to quickly explore diverse paths detecting interesting behaviors early. Another popular strategy is *random path selection*, that has been refined in several variants. For instance, KLEE [Cadar et al. 2008] assigns probabilities to paths based on their length and on the branch arity: it favors paths that have been explored fewer times, preventing starvation caused by loops and other path explosion factors.

Several works, such as EXE [Cadar et al. 2006], KLEE [Cadar et al. 2008], MAYHEM [Cha et al. 2012], and S^{2}E [Chipounov et al. 2012], have discussed heuristics aimed at maximizing code coverage. For instance, the *coverage optimize search* discussed in KLEE [Cadar et al. 2008] computes for each state a weight, which is later used to randomly select states. The weight is obtained by considering how far the nearest uncovered instruction is, whether new code was recently covered by the state, and the state's call stack. Of a similar flavor is the heuristic proposed in [Li et al. 2013], called *subpath-guided search*, which attempts to explore *less traveled* parts of a program by selecting the subpath of the control flow graph that has been explored fewer times. This is achieved by maintaining a frequency distribution of explored subpaths, where a subpath is defined as a consecutive subsequence of length n from a complete path. Interestingly, the value n plays a crucial role with respect to the code coverage achieved by a symbolic engine using this heuristic and no specific value has been shown to be universally optimal. *Shortest-distance symbolic execution* [Ma et al. 2011] does not target coverage, but aims at identifying program inputs that trigger the execution of a specific point in a program. The heuristic is based however, as in coverage-based strategies, on a metric for evaluating the shortest distance to the target point. This is computed as the length of the shortest path in the inter-procedural control flow graph, and paths with the shortest distance are prioritized by the engine.

Other search heuristics try to prioritize paths likely leading to states that are *interesting* according to some goal. For instance, AEG [Avgerinos et al. 2011] introduces two such strategies. The *buggy-path first* strategy picks paths whose past states have contained small but unexploitable bugs. The intuition is that if a path contains some small errors, it is likely that it has not been properly tested. There is thus a good chance that future states may contain interesting, and hopefully exploitable, bugs. Similarly, the *loop exhaustion* strategy explores paths that visit loops. This approach is inspired by the practical observation that common programming mistakes in loops may lead to buffer overflows or other memory-related errors. In order to find exploitable bugs, MAYHEM [Cha et al. 2012] instead gives priority to paths where memory accesses to symbolic addresses are identified or symbolic instruction pointers are detected.

[Zhang et al. 2015] proposes a novel method of dynamic symbolic execution to automatically find a program path satisfying a regular property, i.e., a property (such as file usage or memory safety) that can be represented by a Finite State Machine (FSM). Dynamic symbolic execution is guided by the FSM so that branches of an execution path that are most likely to satisfy the property are explored first. The approach exploits both static and dynamic analysis to compute the priority of a path to be selected for exploration: the states of the FSM that the current execution path has already reached are computed dynamically during the symbolic execution, while backward data-flow analysis is used to compute the future states statically. If the intersection of these two sets is non-empty, there is likely a path satisfying the property.

Fitness functions have been largely used in the context of search-based test generation [McMinn 2004]. A fitness function measures how close an explored path is to achieve the target test coverage. Several works, e.g., [Xie et al. 2009; Cadar and Sen 2013], have applied this idea in the context of symbolic execution. As an example, [Xie et al. 2009] introduces *fitnex*, a strategy for flipping branches in concolic execution that prioritizes paths likely *closer* to take a specific branch. In more detail, given a target branch with an associated condition of the form $|a - c| == 0$, the closeness of a path is computed as $|a - c|$ by leveraging the concrete values of variables a and c in that path. Similar fitness values can be computed for other kinds of branch conditions. The path with the lowest fitness value for a branch is selected by the symbolic engine. Paths that have not reached the branch yet get the worst-case fitness value.

2.4. Symbolic Backward Execution

Symbolic backward execution (SBE) [Chandra et al. 2009; Dinges and Agha 2014b] is a variant of symbolic execution in which the exploration proceeds from a target point to an entry point of a program. The analysis is thus performed in the reverse direction than in canonical (forward) symbolic execution. The main purpose of this approach is typically to identify a test input instance that can trigger the execution of a specific line of code (e.g., an `assert` or `throw` statement). This can very useful for a developer when performing debugging or regression testing over a program. As the exploration starts from the target, path constraints are collected along the branches met during the traversal. Multiple paths can be explored at a time by an SBE engine and, akin to forward symbolic execution, paths are periodically checked for feasibility. When a path condition is proved unsatisfiable, the engine discards the path and backtracks.

[Ma et al. 2011] discusses a variant of SBE dubbed *call-chain backward symbolic execution* (CCBSE). The technique starts by determining a valid path in the function where the target line is located. When a path is found, the engine moves to one of the callers of the function that contains the target point and tries to reconstruct a valid path from the entry point of the caller to the target point. The process is recursively repeated until a valid path from the main function of the program has been reconstructed. The main difference with respect to the traditional SBE is that, although CCBSE follows the call-chain backwards from the target point, inside each function the exploration is done as in traditional symbolic execution.

A crucial requirement for the reversed exploration in SBE, as well as in CCBSE, is the availability of the inter-procedural control flow graph which provides a whole-program control flow and makes it possible to determine the call sites for the functions that are involved in the exploration. Unfortunately, constructing such a graph can be quite challenging in practice. Moreover, a function may have many possible call sites, making the exploration performed by a SBE still very expensive. On the other hand, some practical advantages can arise when the constraints are collected in the reverse direction. We will further discuss these benefits in Section 6.

3. MEMORY MODEL

Our warm-up example of Section 1.1 presented a simplified memory model where data are stored in scalar variables only, with no indirection. A crucial aspect of symbolic execution is how memory should be modeled to support programs with pointers and arrays. This requires extending our notion of memory store by mapping not only variables, but also memory addresses to symbolic expressions or concrete values. In gen-

```
1.  void foobar(unsigned i, unsigned j) {
2.      int a[2] = { 0 };
3.      if (i>1 || j>1) return;
4.      a[i] = 5;
5.      assert(a[j] != 5);
6.  }
```

Fig. 5: Memory modeling example: which values of `i` and `j` make the `assert` fail?

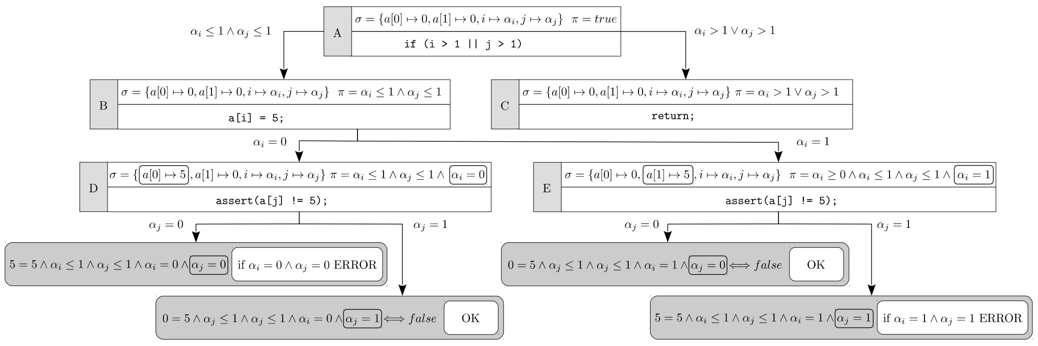

A
$\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = true$
if (i > 1 || j > 1)
$\alpha_i \leq 1 \wedge \alpha_j \leq 1$
$\alpha_i > 1 \vee \alpha_j > 1$
B
$\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i \leq 1 \wedge \alpha_j \leq 1$
a[i] = 5;
C
$\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i > 1 \vee \alpha_j > 1$
return;
$\alpha_i = 0$
$\alpha_i = 1$
D
$\sigma = \{\boxed{a[0] \mapsto 5}, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i \leq 1 \wedge \alpha_j \leq 1 \wedge \boxed{\alpha_i = 0}$
assert(a[j] != 5);
E
$\sigma = \{a[0] \mapsto 0, \boxed{a[1] \mapsto 5}, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i \geq 0 \wedge \alpha_i \leq 1 \wedge \alpha_j \leq 1 \wedge \boxed{\alpha_i = 1}$
assert(a[j] != 5);
$\alpha_j = 0$
$\alpha_j = 1$
$\alpha_j = 0$
$\alpha_j = 1$
$5 = 5 \wedge \alpha_i \leq 1 \wedge \alpha_j \leq 1 \wedge \alpha_i = 0 \wedge \boxed{\alpha_j = 0}$
if $\alpha_i = 0 \wedge \alpha_j = 0$ ERROR
$0 = 5 \wedge \alpha_j \leq 1 \wedge \alpha_j \leq 1 \wedge \alpha_i = 1 \wedge \boxed{\alpha_j = 0} \iff false$
OK
$0 = 5 \wedge \alpha_j \leq 1 \wedge \alpha_j \leq 1 \wedge \alpha_i = 0 \wedge \boxed{\alpha_j = 1} \iff false$
OK
$5 = 5 \wedge \alpha_i \leq 1 \wedge \alpha_j \leq 1 \wedge \alpha_i = 1 \wedge \boxed{\alpha_j = 1}$
if $\alpha_i = 1 \wedge \alpha_j = 1$ ERROR

Fig. 6: Fully symbolic memory via state forking for the example of Figure 5.

eral, a store σ that explicitly models memory addresses can be thought as a mapping that associates memory addresses (indexes) with either expressions over concrete values or symbolic values. We can still support variables by using their address rather than their name in the mapping. In the following, when we write $x \mapsto e$ for a variable x and an expression e we mean $\&x \mapsto e$, where $\&x$ is the concrete address of variable x. Also, if v is an array and c is an integer constant, by $v[c] \mapsto e$ we mean $\&v + c \mapsto e$.

A memory model is an important design choice for a symbolic engine, as it can significantly affect the coverage achieved by the exploration and the scalability of constraint solving [Cadar and Sen 2013]. The *symbolic memory address* problem [Schwartz et al. 2010] arises when the address referenced in the operation is a symbolic expression. In the remainder of this section, we discuss a number of popular solutions.

3.1. Fully Symbolic Memory

At the highest level of generality, an engine may treat memory addresses as fully symbolic. This is the approach taken by a number of works (e.g., BITBLAZE [Song et al. 2008], [Thakur et al. 2010], BAP [Brumley et al. 2011], and [Trtik and Strejček 2014]). Two fundamental approaches, pioneered by King in a seminal paper [King 1976], are the following:

— *State forking.* If an operation reads from or writes to a symbolic address, the state is forked by considering all possible states that may result from the operation. The path constraints are updated accordingly for each forked state.

Example. Consider the code shown in Figure 5. The write operation at line 4 affects either $a[0]$ or $a[1]$, depending on the unknown value of array index i. State forking creates two states after executing the memory assignment to explicitly consider both possible scenarios (Figure 6). The path constraints for the forked states encode the assumption made on the value of i. Similarly, the memory read operation `a[j]` at line 5 may access either $a[0]$ or $a[1]$, depending on the unknown value of array index j. Therefore, for each of the two possible outcomes of the assignment `a[i]=5`, there are two possible outcomes of the `assert`, which are explicitly explored by forking the corresponding states.

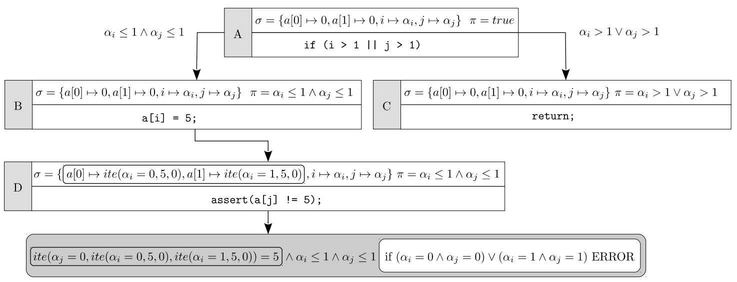

A
$\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = true$
if (i > 1 || j > 1)
$\alpha_i \leq 1 \wedge \alpha_j \leq 1$
$\alpha_i > 1 \vee \alpha_j > 1$
B
$\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i \leq 1 \wedge \alpha_j \leq 1$
a[i] = 5;
C
$\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i > 1 \vee \alpha_j > 1$
return;
D
$\sigma = \{a[0] \mapsto ite(\alpha_i = 0, 5, 0), a[1] \mapsto ite(\alpha_i = 1, 5, 0), i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i \leq 1 \wedge \alpha_j \leq 1$
assert(a[j] != 5);
$ite(\alpha_j = 0, ite(\alpha_i = 0, 5, 0), ite(\alpha_i = 1, 5, 0)) = 5 \wedge \alpha_i \leq 1 \wedge \alpha_j \leq 1$
if $(\alpha_i = 0 \wedge \alpha_j = 0) \vee (\alpha_i = 1 \wedge \alpha_j = 1)$ ERROR

Fig. 7: Fully symbolic memory via if-then-else formulas for the example of Figure 5.

— *if-then-else formulas.* An alternative approach consists in encoding the uncertainty on the possible values of a symbolic pointer into the expressions kept in the symbolic store and in the path constraints, without forking any new states. The key idea is to exploit the capability of some solvers to reason on formulas that contain if-then-else expressions of the form $ite(\mathtt{c}, \mathtt{t}, \mathtt{f})$, which yields `t` if `c` is true, and `f` otherwise. The approach works differently for memory read and write operations. Let α be a symbolic address that may assume the concrete values $a_1, a_2, \ldots$:
  — reading from α yields the expression $ite(\alpha = a_1, \sigma(a_1), ite(\alpha = a_2, \sigma(a_2), \ldots))$;
  — writing an expression e at α updates the symbolic store for each $a_1, a_2, \ldots$ as $\sigma(a_i) \leftarrow ite(\alpha = a_i, e, \sigma(a_i))$.

  Notice that in both cases, a memory operation introduces in the store as many ite expressions as the number of possible values the accessed symbolic address may assume. The ite approach to symbolic memory is used, e.g., in ANGR [Shoshitaishvili et al. 2016] (Section 3.3).

Example. Consider again the example shown in Figure 5. Rather than forking the state after the operation `a[i]=5` at line 4, the if-then-else approach updates the memory store by encoding both possible outcomes of the assignment, i.e., $a[0] \mapsto ite(\alpha_i = 0, 5, 0)$ and $a[1] \mapsto ite(\alpha_i = 1, 5, 0)$ (Figure 7). Similarly, rather than creating a new state for each possible distinct address of `a[j]` at line 5, the uncertainty on j is encoded in the single expression $ite(\alpha_j = 0, \sigma(a[0]), \sigma(a[1])) = ite(\alpha_j = 0, ite(\alpha_i = 0, 5, 0), ite(\alpha_i = 1, 5, 0))$.

An extensive line of research (e.g., EXE [Cadar et al. 2006], KLEE [Cadar et al. 2008], SAGE [Elkarablieh et al. 2009]) leverages the expressive power of some SMT solvers to model fully symbolic pointers. Using a *theory of arrays* [Ganesh and Dill 2007], array operations can in fact be expressed as first-class entities in constraint formulas.

Due to its generality, fully symbolic memory supports the most accurate description of the memory behavior of a program, accounting for all possible memory manipulations. In many practical scenarios, the set of possible addresses a memory operation may reference is small [Song et al. 2008] as in the example shown in Figure 5 where indexes i and j range in a bounded interval, allowing accurate analyses using a reasonable amount of resources. In general, however, a symbolic address may reference any cell in memory, leading to an intractable explosion in the number of possible states. For this reason, a number of techniques have been designed to improve scalability, which elaborate along the following main lines:

— *Representing memory in a compact form.* This approach was taken in [Coppa et al. 2017], which maps symbolic – rather than concrete – address expressions to data, representing the possible alternative states resulting from referencing memory using symbolic addresses in a compact, implicit form. Queries are offloaded to efficient paged interval tree implementations to determine which stored data are possibly referenced by a memory read operation.

— *Trading soundness for performance.* The idea, discussed in the remainder of this section, consists in corseting symbolic exploration to a subset of the execution states by replacing symbolic pointers with concrete addresses.

— *Heap modeling.* An additional idea is to corset the exploration to states where pointers are restricted to be either null, or point to previously heap-allocated objects, rather than to any generic memory location (Section 3.2 and Section 3.4).

3.2. Address Concretization

In all cases where the combinatorial complexity of the analysis explodes as pointer values cannot be bounded to sufficiently small ranges, *address concretization*, which consists in concretizing a pointer to a single specific address, is a popular alternative. This can reduce the number of states and the complexity of the formulas fed to the solver and thus improve running time, although may cause the engine to miss paths that, for instance, depend on specific values for some pointers.

Concretization naturally arises in offline executors (Section 2.2). Prominent examples are DART [Godefroid et al. 2005] and CUTE [Sen et al. 2005], which handle memory initialization by concretizing a reference of type `T*` either to `NULL`, or to the address of a newly allocated object of `sizeof(T)` bytes. DART makes the choice randomly, while CUTE first tries `NULL`, and then, in a subsequent execution, a concrete address. If `T` is a structure, the same concretization approach is recursively applied to all fields of a pointed object. Since memory addresses (e.g., returned by `malloc`) may non-deterministically change at different concrete executions, CUTE uses *logical addresses* in symbolic formulas to maintain consistency across different runs. Another reason for concretization is due to efficiency in constraint solving: for instance, CUTE reasons only about pointer equality constraints using an equivalence graph, resorting to concretization for more general constraints that would need costly SMT theories.

3.3. Partial Memory Modeling

To mitigate the scalability problems of fully symbolic memory and the loss of soundness of memory concretization, MAYHEM [Cha et al. 2012] explores a middle point in the spectrum by introducing a *partial* memory model. The key idea is that written addresses are always concretized and read addresses are modeled symbolically if the contiguous interval of possible values they may assume is small enough. This model is based on a trade-off: it uses more expressive formulas than concretization, since it encodes multiple pointer values per state, but does not attempt to encode all of them like in fully symbolic memory [Avgerinos 2014]. A basic approach to bound the set of possible values that an address may assume consists in trying different concrete values and checking whether they satisfy the current path constraints, excluding large portions of the address space at each trial until a tight range is found. This algorithm comes with a number of caveats: for instance, querying the solver on each symbolic dereference is expensive, the memory range may not be continuous, and the values within the memory region of a symbolic pointer might have structure. MAYHEM thus performs a number of optimizations such as *value-set analysis* [Duesterwald 2004] and forms of query caching (Section 6) to refine ranges efficiently. If at the end of the process the range size exceeds a given threshold (e.g., 1024), the address is concretized. ANGR [Shoshitaishvili et al. 2016] also adopts the partial memory model idea and extends it by optionally supporting write operations on symbolic pointers that range within small contiguous intervals (up to 128 addresses).

```
    class Node {
     protected int value;
     protected Node next;

     // precondition: "acyclic structure"
     public Node add(Node l, int v) {
A|    if (l == null) {
B|     l = new Node();
 |     l.value = v; return l;
      }
C|    l.next = add(l.next, v);
D|    return l;
     }
    }
```

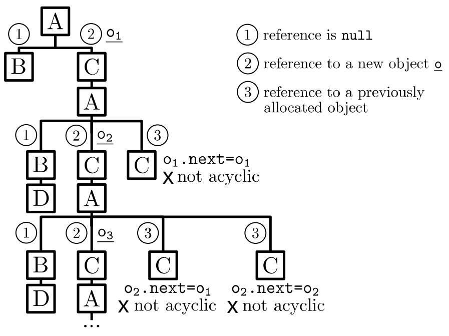


Fig. 8: Example of lazy initialization

3.4. Lazy Initialization

[Khurshid et al. 2003] proposes symbolic execution techniques for advanced object-oriented language constructs, such as those offered by C++ and Java. The authors describe a framework for software verification that combines symbolic execution and model checking to handle linked data structures such as lists and trees.

In particular, they generalize symbolic execution by introducing *lazy initialization* to effectively handle dynamically allocated objects. Compared to our warm-up example from Section 1.1, the state representation is extended with a *heap configuration* used to maintain such objects. Symbolic execution of a method taking complex objects as inputs starts with uninitialized fields, and assigns values to them in a lazy fashion, i.e., they are initialized when first accessed during execution.

When an uninitialized reference field is accessed, the algorithm forks the current state with three different heap configurations, in which the field is initialized with: (1) `null`, (2) a reference to a new object with all symbolic attributes, and (3) a previously introduced concrete object of the desired type, respectively.

[Khurshid et al. 2003; Visser et al. 2004] combine lazy initialization with user-provided *method preconditions*, i.e., conditions that are assumed to be true before the execution of a method. Preconditions are used to characterize those program input states in which the method is expected to behave as intended by the programmer. For instance, we expect a binary tree data structure to be acyclic and with every node - except for the root - having exactly one parent. Conservative preconditions are used to ensure that incorrect heap configurations are eliminated during initialization, speeding up the symbolic execution process.

Example. Figure 8 shows a recursive Java method `add`, which appends a node of type `Node` to a linked list, and a minimal representation of its symbolic execution when applying lazy initialization. The tree nodes represent executions of straight-line fragments of `add`. Initially, fragment A evaluates reference `l`, which is symbolic and thus uninitialized. The symbolic engine considers three options: (1) `l` is `null`, (2) `l` points to a new object, and (3) `l` points to a previously allocated object. Since this is the first time that a reference of type `Node` is met, option (3) is ruled out. The two remaining options are then expanded, executing the involved fragments. While the first path ends after executing fragment B, the second one implicitly creates a new object o_1 due to lazy initialization and then executes C, recursively invoking `add`. When expanding the recursive call, fragment A is executed and the three options are again considered by the engine, which forks into three distinct paths. Option (3) is now taken into account since a `Node` object has been previously allocated (i.e., o_1). However, this path is soon aborted by the engine since it violates the acyclicity precondition (expressed as a comment in this example). The other forked paths are further expanded, repeating the same process. Since the linked list has an unknown maximum length, the exploration can proceed indefinitely. For this reason, it is common to assume an upper bound on the depth of the materialization (i.e., field instantiation) chain.

Recent advances in the area have focused on improving efficiency in generating heap configurations. For instance, in [Deng et al. 2012] the concretization of a reference variable is deferred until the object is actually accessed. The work also provides a formalization of lazy initialization. [Rosner et al. 2015] instead employs bound refinement to prune uninteresting heap configurations by using information from already concretized fields, while a SAT solver is used to check whether declarative – rather than imperative as in the original algorithm – preconditions hold for a given configuration.

4. INTERACTION WITH THE ENVIRONMENT

As most programs are not self-contained, a symbolic engine has to take into account their frequent interactions with the surrounding software stack. A typical example is data flows that take place through features of the underlying operating system (e.g., file system, environment variables, network). Functions controlled by the system environment are often referred to as *external*. Modern applications pose further challenges when they interact with the user via other components (e.g., Swing, Android), or invoke special features of their execution runtimes. Missing symbolic data flows through these software elements might indeed affect the meaningfulness of the analysis.

System Environment. A body of early works (e.g., DART [Godefroid et al. 2005], CUTE [Sen et al. 2005], and EXE [Cadar et al. 2006]) include the system environment in the analysis by actually executing external calls using concrete arguments for them. This indeed limits the behaviors they can explore compared to a fully symbolic strategy, which on the other hand might be unfeasible. In an online executor this choice may also result in having external calls from distinct paths of execution interfere with each other. As there is no mechanism for tracking the side effects of each external call, there is potentially a risk of state inconsistency, e.g., an execution path may read from a file while at the same time another execution path is trying to delete it.

A way to overcome this problem is to create abstract models that capture these interactions. For instance, in KLEE [Cadar et al. 2008] symbolic files are supported through a basic *symbolic file system* for each execution state, consisting of a directory with n symbolic files whose number and sizes are specified by the user. An operation on a symbolic file results in forking $n+1$ state branches: one for each possible file, plus an optional one to capture unexpected errors in the operation. As the number of functions in a standard library is typically large and writing models for them is expensive and error-prone [Ball et al. 2006], models are generally implemented at system call-level rather than library level. This enables the symbolic exploration of the libraries as well.

AEG [Avgerinos et al. 2011] models most of the system environment that could be used by an attacker as input source, including the file system, network sockets, and environment variables. Additionally, more than 70 library and system calls are emulated, including thread- and process-related system calls, and common formatting functions to capture potential buffer overflows. Symbolic files are handled as in KLEE [Cadar et al. 2008], while symbolic sockets are dealt with in a similar manner, with packets and their payloads being processed as in symbolic files and their contents. CLOUD9 [Bucur et al. 2011] supports additional POSIX libraries, and allows users to control advanced conditions in the testing environment. For instance, it can simulate reordering, delays, and packet dropping caused by a fragmented network data stream.

S^{2}E [Chipounov et al. 2012] remarks that models, other than expensive to write, rarely achieve full accuracy, and may quickly become stale if the modeled system changes. It would thus be preferable to let analyzed programs interact with the real environment while exploring multiple paths. However, this must be done without incurring in environment interferences or state inconsistencies. To achieve this goal, S^{2}E resorts to virtualization to prevent propagation of side effects across independent execution paths when interacting with the real environment. QEMU [Bellard 2005] is used to emulate the full software stack: instructions are transparently translated into micro operations run by the native host, while an x86-to-LLVM lifter is used to perform symbolic execution of the instructions sequence in KLEE [Cadar et al. 2008]. This allows S^{2}E to properly evaluate any side-effects due to the environment. Notice that whenever a symbolic branch condition is evaluated, the execution engine forks a parallel instance of the emulator to explore the alternative path. Selective symbolic execution (Section 2.1) is used to limit the scope of symbolic exploration across the software stack, partially mitigating the overhead of emulating a full stack (e.g., user code, libraries, drivers) that can significantly limit the scalability of the overall solution.

DART's approach [Godefroid et al. 2005] is different, as the goal is to enable automated unit testing. DART deems as foreign interfaces all the external variables and functions referenced in a C program along with the arguments for a top-level function. External functions are simulated by nondeterministically returning any value of their specified return type. To allow the symbolic exploration of library functions that do not depend on the environment, the user can adjust the boundary between external and non-external functions to tune the scope of the symbolic analysis.

Application Environment. We now discuss possible solutions for dealing with software elements that carry out control and data flows on the behalf of the program under analysis. Instances of this problem arise for instance in frameworks like Swing and Android, which embody abstract designs to invoke application code (e.g., via callbacks) during user interaction [Jeon et al. 2016]. Symbolic values flow outside the boundaries of the analysis also for applications running in managed runtimes, e.g., when calling native Java methods or unmanaged code in .NET [Anand et al. 2007]. Such features complicate the implementation of an engine: for instance, native methods and reflection in Java depend on the internals of the underlying JVM [Anand 2012]. Closed-source components might represent another instance of this problem.

Similarly as in system environment modeling, early works such as DART [Godefroid et al. 2005] and CUTE [Sen et al. 2005] deal with calls to other software components by executing them with concrete arguments. This may result in an incomplete exploration, failing to generate test inputs for feasible program paths. On the other hand, a symbolic execution of their code is unlikely to succeed for a number of reasons: for instance, the implementation of externally simple behaviors is often complex as it has to allow for extensibility and maintainability, or may contain details irrelevant to the exploration, such as how to display a button that triggers a callback [Jeon et al. 2016]. One solution would be to mimic external components with simpler and more abstract models. However, writing component models manually – which can be a daunting task per se – might be hard due to the unavailability of the source code, and applications using unsupported models would remain out of reach.

Some works (e.g., [Anand et al. 2007; Xiao et al. 2011]) explore techniques to pinpoint which entities from a component may hold symbolic values in a symbolic exploration, and thus require human intervention (e.g., writing a model) for their analysis. A different line of research has instead attempted to generate models automatically, which may be the only viable option for closed-source components. [Ceccarello and Tkachuk 2014; van der Merwe et al. 2015] employ program slicing to extract the code that manipulates a given set of fields relevant for the analysis, and build abstract models from it. [Jeon et al. 2016] takes a step further by using program synthesis to produce models for Java frameworks. Such models provide equivalent instantiations of design patterns that are heavily used in many frameworks: this helps symbolic executors discover control flow – such as callbacks to user code through an observer pattern – that would otherwise be missed. An advantage of using program synthesis is that it can generate more concise models than slicing, as it abstracts away the details and entanglements of how a program is written by capturing its functional behavior.

5. PATH EXPLOSION

One of the main challenges of symbolic execution is the path explosion problem: a symbolic executor may fork off a new state at every branch of the program, and the total number of states may easily become exponential in the number of branches. Keeping track of a large number of pending branches to be explored, in turn, impacts both the running time and the space requirements of the symbolic executor.

The main sources of path explosion are loops and function calls. Each iteration of a loop can be seen as an `if-goto` statement, leading to a conditional branch in the execution tree. If the loop condition involves one or more symbolic values, the number of generated branches may be potentially infinite, as suggested by the following example.

Example. Consider the following code fragment [Cadar and Sen 2013]:

```
int x = sym_input(); // e.g., read from file
while (x > 0) x = sym_input();
```

where `sym_input()` is an external routine that interacts with the environment (e.g., by reading input data from a network) and returns a fresh symbolic input. The path constraint set at any final state has the form:

$$\pi = \left(\bigwedge_{i \in [1,k]} \alpha_i > 0 \right) \wedge (\alpha_{k+1} \leq 0)$$

where k is the number of iterations and α_i is the symbol produced by `sym_input()` at the i-th iteration.

While it would be simple (and is indeed common) to bound the loop exploration up to a limited number of iterations, interesting paths could be easily missed with this approach. A large number of works have thus explored more advanced strategies, e.g., by characterizing similarities across distinct loop iterations or function invocations through summarization strategies that prevent repeated explorations of a code portion or by inferring invariants that inductively describe the properties of a computation. In the remainder of this section we present a variety of prominent techniques, often based on the computation of an under-approximation of the analysis with the aim of exploring only a relevant subset of the state space.

5.1. Pruning Unrealizable Paths

A first natural strategy to reduce the path space is to invoke the constraint solver at each branch, pruning unrealizable branches: if the solver can prove that the logical formula given by the path constraints of a branch is not satisfiable, then no assignment of the program input values could drive a real execution toward that path, which can be safely discarded by the symbolic engine without affecting soundness. An example of this strategy is provided in Figure 9.

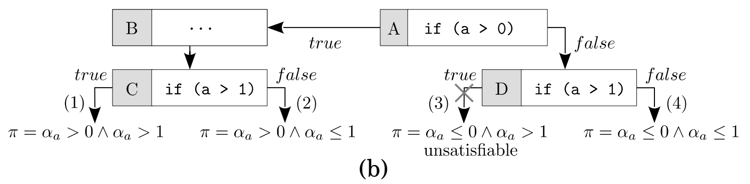
This approach is commonly referred to as *eager evaluation* of path constraints, since constraints are eagerly checked at each branch, and is typically the default in most symbolic engines. We refer to Section 6 for a discussion of the opposite strategy, called *lazy evaluation*, aimed at reducing the burden on the constraint solver.

An orthogonal approach that can help reduce the number of paths to check is presented in [Schwartz-Narbonne et al. 2015]. While an SMT solver can be used to explore a large search space one path at a time, it will often end up reasoning over control

```
if (a > 0) { ... }

if (a > 1) { ... }
```

(a)


B
...
true
A
if (a > 0)
false
true
C
if (a > 1)
false
(1)
(2)
$\pi = \alpha_a > 0 \wedge \alpha_a > 1$
$\pi = \alpha_a > 0 \wedge \alpha_a \leq 1$
true
D
if (a > 1)
false
(3)
(4)
$\pi = \alpha_a \leq 0 \wedge \alpha_a > 1$
unsatisfiable
$\pi = \alpha_a \leq 0 \wedge \alpha_a \leq 1$

(b)

Fig. 9: Pruning unrealizable paths example: (a) code fragment; (b) symbolic execution of the code fragment: the *true* branch at node D is not explored since its path constraints ($\alpha_a \leq 0 \wedge \alpha_a > 1$) are not satisfiable.

flows shared by many paths. The work exploits this observation by extracting a minimal *unsat core* from each path that is proved to be unsatisfiable, removing as many statements as possible while preserving unsatisfiability. An engine could thus exploit unsat cores to discard paths that share the same (unsatisfiable) statements.

5.2. Function and Loop Summarization

When a code fragment – be it a function or a loop body – is traversed several times, the symbolic executor can build a summary of its execution for subsequent reuse.

Function Summaries. A function f may be called multiple times throughout the execution, either at the same calling context or at different ones. Differently from plain executors, which would execute f symbolically at each invocation, the compositional approach proposed in [Godefroid 2007] for concolic executors dynamically generates *function summaries*, allowing the executor to effectively reuse prior discovered analysis results. The technique captures the effects of a function invocation with a formula ϕ_w that conjoins constraints on the function inputs observed during the exploration of a path w, describing equivalence classes of concrete executions, with constraints observed on the outputs. Inputs and outputs are defined in terms of accessed memory locations. A function summary is a propositional logic formula defined as the disjunction of ϕ_w formulas from distinct classes, and feasible inter-procedural paths are modeled by composing symbolic executions of intra-procedural ones. [Anand et al. 2008] extends compositional symbolic execution by generating summaries as first-order logic formulas with uninterpreted functions, allowing the formation of incomplete summaries (i.e., capturing only a subset of the paths within a function) that can be expanded on demand during the inter-procedural analysis as more statements get covered.

[Boonstoppel et al. 2008] explores a different flavor of summarization, based on the following intuition: if two states differ only for some program values that are not read later, the executions generated by the two states will produce the same side effects. Side effects of a code fragment can be therefore cached and possibly reused later.

Loop Summaries. Akin to function calls, partial summarizations for loops can be obtained as described in [Godefroid and Luchaup 2011]. A loop summary uses pre- and post-conditions that are dynamically computed during the symbolic execution by reasoning on the dependencies among loop conditions and symbolic variables. Caching loop summaries not only allows the symbolic engine to avoid redundant executions of the same loop in the same program state, but makes it also possible to generalize the summary to cover different executions of the same loop under different conditions.

Early works can generate summaries only for loops that update symbolic variables across iterations by adding a fixed amount to them. Also, they cannot handle nested loops or *multi-path loops*, i.e., loops with branches within their body. Proteus [Xie et al. 2016] is a general framework proposed for summarizing multi-path loops. It classifies loops according to the patterns of values changes in path conditions (i.e., whether an induction variable is updated) and of the interleaving of paths within the loop (i.e., whether there is a regularity). The classification leverages an extended form of control flow graph, which is then used to construct an automata that models the interleaving. The automata is traversed in a depth-first fashion and a disjunctive summarization is constructed for all the feasible traces in it, where a trace represents an execution in the loop. The classification determines if a loop can be captured either precisely or approximately (which can still be of practical relevance), or it cannot. Precise summarization of multi-path loops with irregular patterns or non-inductive updates, and more importantly summarization of nested loops remain open research problems.

Of a different flavor is the compaction technique introduced in [Slaby et al. 2013], wherethe analysis of cyclic paths in the control flow graph yields *templates* that declaratively describe the program states generated by a portion of code as a *compact* symbolic execution tree. By exploiting templates, the symbolic execution engine can explore a significantly reduced number of program states. A drawback of this approach is that templates introduce quantifiers in the path constraints: in turn, this may significantly increase the burden on the constraint solver.

5.3. Path Subsumption and Equivalence

A large symbolic state space offers scope for techniques that explore path similarity to, e.g., discard paths that cannot lead to new findings, or abstract away differences when profitable. In this section we discuss a number of works along these lines.

Interpolation. Modern SAT solvers rely on a mutual reinforcing combination of search and deduction, using the latter to drive the former away from a conflict when it becomes blocked. In a similar manner, symbolic execution can benefit from *interpolation* techniques to derive properties from program paths that did not show a desired property, so to prevent the exploration of similar paths that would not satisfy it either.

Craig interpolants [Craig 1957] allow deciding what information about a formula is relevant to a property. Assuming an implication $P \rightarrow Q$ holds in some logic, one can construct an interpolant I such that $P \rightarrow I$ and $I \rightarrow Q$ are valid, and every non-logical symbol in I occurs in both P and Q. Interpolation is commonly used in program verification as follows: given a refutation proof for an unsatisfiable formula $P \wedge Q$, a *reverse interpolant* I can be constructed such that $P \rightarrow I$ is valid and $I \wedge Q$ is unsatisfiable.

Interpolation has largely been employed in model checking, predicate abstraction, predicate refinement, theorem proving, and other areas. For instance, interpolants provide a methodology to extend *bounded model checking* – which aims at falsifying safety properties of a program for which the transition relation is unrolled up to a given bound – to the unbounded case. In particular, since bounded proofs often contain the ingredients of unbounded proofs, interpolation can help construct an over-approximation of all reachable final states from the refutation proof for the bounded case, obtaining an over-approximation that is strong enough to prove absence of violations.

Subsumption with Interpolation. Interpolation can be used to tackle the path explosion problem when symbolically verifying programs marked (e.g., using assertions) with explicit error locations. As the exploration proceeds, the engine annotates each program location with conditions summarizing previous paths through it that have failed to reach an error location. Every time a branch is encountered, the executor checks whether the path conditions are subsumed by the previous explorations. In a best-case scenario, this approach can reduce the number of visited paths exponentially.

[McMillan 2010] proposes an annotation algorithm for branches and statements such that if their labels are implied by the current state, they cannot lead to an error location. Interpolation is used to construct weak labels that allow for an efficient computation of implication. [Yi et al. 2015] proposes a similar redundancy removal method called *postconditioned symbolic execution*, where program locations are annotated with a postcondition, i.e., the *weakest precondition* summarizing path suffixes from previous explorations. The intuition here is that the weaker the interpolant is, the more likely it would enable path subsumption. Postconditions are constructed incrementally from fully explored paths and propagated backwards. When a branch is encountered, the corresponding postcondition is negated and added to the path constraints, which become unsatisfiable if the path is subsumed by previous explorations.

The soundness of path subsumption relies on the fact that an interpolant computed for a location captures the entirety of paths going through it. Thus, the path selection strategy plays a key role in enabling interpolant construction: for instance, DFS is very convenient as it allows exploring paths in full quickly, so that interpolants can be constructed and eventually propagated backwards; BFS instead hinders subsumption as interpolants may not available when checking for redundancy at branches as similar paths have not been explored in full yet. [Jaffar et al. 2013] proposes a novel strategy called *greedy confirmation* that decouples the path selection problem from the interpolant formation, allowing users to benefit from path subsumption when using heuristics other than DFS. Greedy confirmation distinguishes betweens nodes whose trees of paths have been explored in full or partially: for the latter, it performs limited traversal of additional paths to enable interpolant formation.

Interpolation has been proven to be useful for allowing the exploration of larger portions of a complex program within a given time budget. [Yi et al. 2015] claims that path redundancy is abundant and widespread in real-world applications. Typically, the overhead of interpolation - which can be performed within the SMT solver or in a dedicated engine - slows down the exploration in the early stages, then its benefits eventually start to pay off, allowing for a much faster exploration [Jaffar et al. 2013].

Unbounded Loops. The presence of an unbounded loop in the code makes it harder to perform sound subsumption at program locations in it, as a very large number of paths can go through them. [McMillan 2010] devises an iterative deepening strategy that unrolls loops until a fixed depth and tries to compute interpolants that are *loop invariant*, so that they can be used to prove the unreachability of error nodes in the unbounded case. This method however may not terminate for programs that require disjunctive loop invariants. [Jaffar et al. 2012b] thus proposes a strategy to compute speculative invariants strong enough to make the symbolic execution of the loop converge quickly, but also loose enough to allow for path subsumption whenever possible. In a follow-up work [Jaffar et al. 2012a] loop invariants are discovered separately during the symbolic execution using a widening operator, and weakest preconditions for path subsumption are constructed such that they are entailed by the invariants.

We believe that the idea of using abstract interpretation in this setting – originally suggested in [Jaffar et al. 2009] – deserves further investigation, as it can benefit from its many applications in other program verification techniques, and is amenable to an efficient implementation in mainstream symbolic executors, provided that the constructed invariants are accurate enough to capture the (un)rechability of error nodes.

Subsumption with Abstraction. An approach not based on interpolation is taken in [Anand et al. 2009], which describes a two-fold subsumption checking technique for symbolic states. A symbolic state is defined in terms of a symbolic heap and a set of constraints over scalar variables. The technique thus targets programs that manipulate not only scalar types, but also uninitialized or partially initialized data structures. An algorithm for matching heap configurations through a graph traversal is presented, while an off-the-shelf solver is used to reason about subsumption for scalar data.

To cope with a possibly unbounded number of states, the work proposes abstraction to make the symbolic state space finite and thus subsumption effective. Abstractions can summarize both the heap shape and the constraints on scalar data; examples are given for linked lists and arrays. Subsumption checking happens on under-approximate states, meaning that feasible behaviors could be missed. The authors employ the technique in a falsification scenario in combination with model checking, leaving to future work an application to verification based on symbolic execution only.

Path Partitioning. Dependence analyses for control and data flows expose casual relationships that one can use during the exploration to filter out paths unable to reveal additional program behavior. [Majumdar and Xu 2009] partitions inputs for concolic execution in non-interfering blocks, symbolically exploring each block while others are kept fixed to concrete values. Interference of two inputs happens when they jointly affect one statement, or statements linked by control or data dependences. [Qi et al. 2013] focuses on outputs, placing two paths in the same partition if they have the same relevant slice with respect to the program output. A relevant slice is the transitive closure of dynamic data and control dependencies, and also of potential dependencies involving statements that affect the output by not getting executed. [Wang et al. 2017] explores also faults irrelevant to the output by building relevant slices for individual statements, capturing how they are computed from symbolic inputs. A dependency analysis efficiently checks for equivalence of slices, deeming a path redundant when the slices for all its statement instances are collectively covered by previous paths.

5.4. Under-constrained Symbolic Execution

A possible approach to avoid path explosion is to cut the code to be analyzed, say a function, out of its enclosing system and check it in isolation. Lazy initialization with user-specified preconditions (Section 3.4) follows this principle in order to automatically reconstruct complex data structures. However, taking a code region out of an application may be quite difficult due to the entanglements with the surrounding environment [Engler and Dunbar 2007]: errors detected in a function analyzed in isolation may be false positives, as the input may never assume certain values when the function is executed in the context of a full program. Some prior works, e.g., [Csallner and Smaragdakis 2005], first analyze the code in isolation and then test the generated crashing inputs using concrete executions to filter out false positives.

Under-constrained symbolic execution [Engler and Dunbar 2007] is a twist on symbolic execution that allows the analysis of a function in isolation by marking its symbolic inputs, as well as any global data that may affect its execution, as *under-constrained*. Intuitively, a symbolic variable is under-constrained when in the analysis we do not account for constraints on its value that should have been collected along the path prefix from the program's entry point to the function. In practice, a symbolic engine can automatically mark data as under-constrained without manual intervention by tracing memory accesses and identifying their location: e.g., a function's input can be detected when a memory read is performed on uninitialized data located on the stack. Under-constrained variables have the same semantics as classical fully constrained symbolic variables except when used in an expression that can yield an error. In particular, an error is reported only if all the solutions for the currently known constraints on the variable cause it to occur, i.e., the error is context-insensitive and thus a true positive. Otherwise, its negation is added to the path constraints and execution resumes as normal. This approach can be regarded as an attempt to reconstruct preconditions from the checks inserted in the code: any subsequent action violating an added negated constraint will be reported as an error. In order to keep this analysis correct, marks must be propagated between variables whenever any expression involves both under- and fully constrained values. For instance, a comparison of the form `a > b`, where `a` is under-constrained and `b` is not, forces the engine to propagate the mark from `a` to `b`, similarly as in taint analysis when handling tainted values. Marks are typically tracked by the symbolic engine using a shadow memory.

Although this technique is not sound as it may miss errors, it can still scale to find interesting bugs in larger programs. Also, the application of under-constrained symbolic execution is not limited to functions only: for instance, if a code region (e.g., a loop) may be troublesome for the symbolic executor, it can be skipped by marking the locations it affects as under-constrained. Since in general it is not easy to understand which data could be affected by the execution of some skipped code, manual annotation may be needed in order to keep the analysis correct.

5.5. Exploiting Preconditions and Input Features

Another way to reduce the path explosion is to leverage knowledge of some input properties. AEG [Avgerinos et al. 2011] proposes *preconditioned symbolic execution* to reduce the number of explored states by directing the exploration to a subset of the input space that satisfies a precondition predicate. The rationale is to focus on inputs that may lead to certain behaviors of the program (e.g., narrowing down the exploration to inputs of maximum size to reveal potential buffer overflows). Preconditioned symbolic execution trades soundness for performance: well-designed preconditions should be neither too specific (they would miss interesting paths) nor too generic (they would compromise the speedups resulting from the space state reduction). Instead of starting from an empty path constraints set, the approach adds the preconditions to the initial π so that the rest of the exploration will skip branches that do not satisfy them. While adding more constraints to π at initialization time is likely to increase the burden on the solver, required to perform a larger number of checks at each branch, this may be largely outweighted by the performance gains due to the smaller state space.

Common types of preconditions considered in symbolic execution are: *known-length* (i.e., the size of a buffer is known), *known-prefix* (i.e., a buffer has a known prefix), and *fully known* (i.e., the content of a buffer is fully concrete). These preconditions are rather natural when dealing with code that operates over inputs with a well-known or predefined structure, such as string parsers or packet processing tools.

Example. Consider the following simplified packet header processing code: `pkt` points to the input buffer, while `header` to the fixed expected content. If no precondition is considered, then this code can generate an exponential number of paths since any mismatch forces a new call to `get_input`. On the other hand, if a *known prefix* precondition is set on the input, then only a single path is generated when exploring the loop. The engine can thus focus its exploration on `parse_payload()`.

```
start: get_input(&pkt);
for(k = 0; k < 128; k++)
  if (pkt[k] != header[k])
    goto start;
parse_payload(&pkt)
```

Of a different flavor is the work by [Saxena et al. 2009], which presents a technique, called *loop-extended symbolic execution*, that is able to effectively explore a loop whenever a grammar describing the input program is available. Relating the number of iterations with features of the program input can profitably guide the exploration of the program states generated by a loop, reducing the path explosion problem.

5.6. State Merging

State merging is a powerful technique that fuses different paths into a single state. A merged state is described by a formula that represents the disjunction of the formulas that would have described the individual states if they were kept separate. Differently from other static program analysis techniques such as abstract interpretation, merging in symbolic execution does not lead to over-approximation.

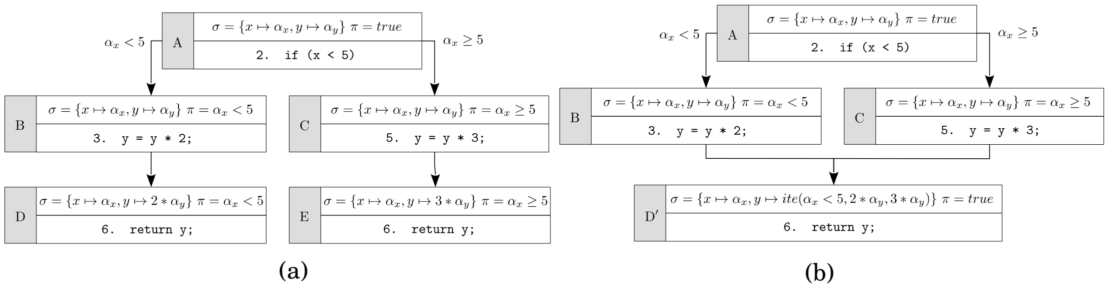

$\alpha_x < 5$
A
$\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$ $\pi = true$
2. if (x < 5)
$\alpha_x \geq 5$
B
$\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$ $\pi = \alpha_x < 5$
3. y = y * 2;
C
$\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$ $\pi = \alpha_x \geq 5$
5. y = y * 3;
D
$\sigma = \{x \mapsto \alpha_x, y \mapsto 2 * \alpha_y\}$ $\pi = \alpha_x < 5$
6. return y;
E
$\sigma = \{x \mapsto \alpha_x, y \mapsto 3 * \alpha_y\}$ $\pi = \alpha_x \geq 5$
6. return y;
(a)
D′
$\sigma = \{x \mapsto \alpha_x, y \mapsto ite(\alpha_x < 5, 2 * \alpha_y, 3 * \alpha_y)\}$ $\pi = true$
6. return y;
(b)

Fig. 10: Symbolic execution of function `foo`: (a) without and (b) with state merging.

Example. Consider function `foo` shown below and its symbolic execution tree shown in Figure 10a. Initially (execution state A) the path constraints are *true* and input arguments `x` and `y` are associated with symbolic values α_x and α_y, respectively.

```
1. void foo(int x, int y) {
2.   if (x < 5)
3.     y = y * 2;
4.   else
5.     y = y * 3;
6.   return y;
7. }
```

After forking due to the conditional branch at line 2, a different statement is evaluated and different assumptions are made on symbol α_x (states B and C, respectively). When the `return` at line 6 is eventually reached on all branches, the symbolic execution tree gets populated with two additional states, D and E. In order to reduce the number of active states, the symbolic engine can perform state merging. For instance, Figure 10b shows the symbolic execution DAG for the same piece of code when a state merging operation is performed before evaluating the `return` at line 6: D' is a merged state that fully captures the former execution states D and E using the *ite* expression $ite(\alpha_x < 5, 2*\alpha_y, 3*\alpha_y)$ (Section 3.1). Note that the path constraints of the execution states D and E can be merged into the disjunction formula $\alpha_x < 5 \vee \alpha_x \geq 5$ and then simplified to *true* in D'.

Tradeoffs: to Merge or Not to Merge? In principle, it may be profitable to apply state merging whenever two symbolic states about to evaluate the same statement are very similar (i.e., differ only for few elements) in their symbolic stores. Given two states $(stmt,\ \sigma_1,\ \pi_1)$ and $(stmt,\ \sigma_2,\ \pi_2)$, the merged state can be constructed as $(stmt,\ \sigma',\ \pi_1 \vee \pi_2)$, where σ' is the merged symbolic store between σ_1 and σ_2 built with *ite* expressions accounting for the differences in storage, while $\pi_1 \vee \pi_2$ is the union of the path constraints from the two merged states. Control-flow structures such as if-else statements (as in the previous example) or simple loops often yield rather similar successor states that represent very good candidates for state merging.

Early works [Godefroid 2007; Hansen et al. 2009] have shown that merging techniques effectively decrease the number of paths to explore, but also put a burden on constraints solvers, which can be hampered by disjunctions. Merging can also introduce new symbolic expressions in the code, e.g., when merging different concrete values from a conditional assignment into a symbolic expression over the condition. [Kuznetsov et al. 2012] provides an excellent discussion of the design space of state merging techniques. At the one end of the spectrum, complete separation of paths used in search-based symbolic execution (Section 2.3) performs no merge. At the other end, static state merging combines states at control-flow join points, essentially representing a whole program with a single formula. Static state merging is used in whole-program verification condition generators [Xie and Aiken 2005; Babic and Hu 2008]), which usually trade precision for scalability e.g., by unrolling loops only once.

Merging Heuristics. Intermediate merging solutions adopt heuristics to identify state merges that can speed the exploration process up. Indeed, generating larger symbolic expressions and possibly extra solvers invocations can outweigh the benefit of having fewer states, leading to poorer overall performance [Hansen et al. 2009; Kuznetsov et al. 2012]. *Query count estimation* [Kuznetsov et al. 2012] relies on a simple static analysis to identify how often each variable is used in branch conditions past any given point in the CFG. The estimate is used as a proxy for the number of solver queries that a given variable is likely to be part of. Two states make a good candidate for merging when their differing variables are expected to appear infrequently in later queries. *Veritesting* [Avgerinos et al. 2014] implements a form of merging heuristic based on a distinction between easy and hard statements, where the latter involve indirect jumps, system calls, and other operations for which precise static analyses are difficult to achieve. Static merging is performed on sequences of easy statements, whose effects are captured using ite expressions, while per-path symbolic exploration is done whenever a hard-to-analyze statement is encountered.

Dynamic State Merging. In order to maximize merging opportunities, a symbolic engine should traverse a CFG so that a combined state for a program point can be computed from its predecessors, e.g., if the graph is acyclic, by following a topological ordering. However, this would prevent search exploration strategies that prioritize "interesting" states. [Kuznetsov et al. 2012] introduces *dynamic state merging* which works regardless of the exploration order imposed by the search strategy. Suppose the symbolic engine maintains a worklist of states and a bounded history of their predecessors. When the engine has to pick the next state to explore, it first checks whether there are two states s_1 and s_2 from the worklist such that they do not match for merging, but s_1 and a predecessor of s_2 do. If the expected similarity between s_2 and a successor of s_1 is also high, the algorithm attempts a merge by advancing the execution of s_1 for a fixed number of steps. This captures the idea that if two states are similar, then also their respective successors are likely to become similar in a few steps. If the merge fails, the algorithm lets the search heuristic pick the next state to explore.

5.7. Leveraging Program Analysis and Optimization Techniques

A deeper understanding of a program's behavior can help a symbolic engine optimize its analysis and focus on promising states, e.g., by pruning uninteresting parts of the computation tree. Several classical program analysis techniques have been explored in the symbolic execution literature. We now briefly discuss some prominent examples.

Program Slicing. This analysis, starting from a subset of a program's behavior, extracts from the program the minimal sequence of instructions that faithfully represents that behavior [Weiser 1984]. This information can help a symbolic engine in several ways: for instance, [Shoshitaishvili et al. 2015] exploits backward program slicing to restrict symbolic exploration toward a specific target program point.

Taint Analysis. This technique [Schwartz et al. 2010] attempts to check which variables of a program may hold values derived from potentially dangerous external sources such as user input. The analysis can be performed both statically and dynamically, with the latter yielding more accurate results. In the context of symbolic execution, taint analysis can help an engine detect which paths depend on tainted values. For instance, [Cha et al. 2012] focuses its analysis on paths where a jump instruction is tainted and uses symbolic execution to generate an exploit.

Fuzzing. This software testing approach randomly mutates user-provided test inputs to cause crashes or assertion failures, possibly finding potential memory leaks. Fuzzing can be augmented with symbolic execution to collect constraints for an input and negate them to generate new inputs. On the other hand, a symbolic executor can be augmented with fuzzing to reach deeper states in the exploration more quickly and efficiently. Two notable embodiments of this idea are represented by *hybrid concolic testing* [Majumdar and Sen 2007] and Driller [Stephens et al. 2016].

Branch Predication. This is a strategy for mitigating misprediction penalties in pipelined executions by avoiding jumps over very small sections of code: for instance, control-flow forking constructs such as the C ternary operator can be replaced with a predicated `select` instruction. [Collingbourne et al. 2011] reports an exponential decrease in the number of paths to explore from the adoption of this strategy when cross-checking two implementations of a program using symbolic execution.

Type Checking. Symbolic analysis can be effectively mixed with typed checking [Khoo et al. 2010]: for instance, type checking can determine the return type of a function that is difficult to analyze symbolically: such information can then potentially be used by the executor to prune certain paths[1].

Program Differencing. Dependence analyses can identify branches and data flows affected by code edits. *Directed incremental symbolic execution* [Yang et al. 2014] statically identifies CFG nodes affected by changes, and uses such information to drive the exploration to only those paths that exercise uncovered sequences of affected nodes.

Compiler Optimizations. [Cadar 2015] argues that program optimization techniques should be a first-class ingredient of practical implementations of symbolic execution, alongside widely accepted solutions such as search heuristics, state merging, and constraint solving optimizations. In fact, program transformations can affect both the complexity of the constraints generated during path exploration and the exploration itself. For instance, precomputing the results of a function using a lookup table leads to a larger number of constraints in the path conditions due to memory accesses, while applying strength reduction for multiplication may result in a chain of addition operations that is more expensive for a constraint solver. Also, the way high-level `switch` statements are compiled can significantly affect the performance of path exploration, while resorting to conditional instructions such as `select` in LLVM or `setcc` and `cmov` in x86 can avoid expensive state forking by yielding simple *ite* expressions instead.

While the effects of a compiler optimization can usually be predicted on the number or size of the instructions executed at run time, a similar reduction is not obvious in symbolic execution [Dong et al. 2015], mostly because the constraint solver is typically used as a black-box. To the best of our knowledge, only a few works have attempted to analyze the impact of compiler optimizations on constraint generation and path exploration [Wagner et al. 2013; Dong et al. 2015], leaving interesting open questions. Of a different flavor is the work presented in [Perry et al. 2017], which explores transformations such as dynamic constant folding and optimized constraint encoding to speed up memory operations in symbolic executors based on theories of arrays (Section 3.1).

6. CONSTRAINT SOLVING

Constraint satisfaction problems arise in many domains, including analysis, testing, and verification of software programs. Constraint solvers are decision procedures for problems expressed in logical formulas: for instance, the boolean satisfiability problem (also known as SAT) aims at determining whether there exists an interpretation of the symbols of a formula that makes it true. Although SAT is a well-known NP-complete problem, recent advances have moved the boundaries for what is intractable when it comes to practical applications [De Moura and Bjørner 2011].

[1]The work also discusses how a symbolic analysis can help type checking, e.g, by providing context-sensitive properties over a variable that would rule out certain type errors, improving the precision of the type checker.

Observe that some problems are more naturally described with languages that are more expressive than the one of boolean formulas with logical connectives. For this reason, satisfiability modulo theories (SMT) generalize the SAT problem with supporting theories to capture formulas involving, for instance, linear arithmetic and operations over arrays. SMT solvers map the atoms in an SMT formula to fresh boolean variables: a SAT decision procedure checks the rewritten formula for satisfiability, and a theory solver checks the model generated by the SAT procedure.

SMT solvers show several distinctive strengths. Their core algorithms are generic, and can handle complex combinations of many individual constraints. They can work incrementally and backtrack as constraints are added or removed, and provide explanations for inconsistencies. Theories can be added and combined in arbitrary ways, e.g., to reason about arrays of strings. Decision procedures do not need to be carried out in isolation: often, they are profitably combined to reduce the amount of time spent in heavier procedures, e.g., by solving linear parts first in a non-linear arithmetic formula. Incomplete procedures are valuable too: complete but expensive procedures get called only when conclusive answers could not be produced. All these factors allows SMT solvers to tackle large problems that no single procedure can solve in isolation[2].

In a symbolic executor, constraint solving plays a crucial role in checking the feasibility of a path, generating assignments to symbolic variables, and verifying assertions. Over the years, different solvers have been employed by symbolic executors, depending on the supported theories and the relative performance at the time. For instance, the STP [Ganesh and Dill 2007] solver has been employed in, e.g., EXE [Cadar et al. 2006], KLEE [Cadar et al. 2008], and AEG [Avgerinos et al. 2011], which all leverage its support for bit-vector and array theories. Other executors such as JAVA PATHFINDER [Pasareanu and Rungta 2010] have complemented SMT solving with additional decision procedures (e.g., libraries for constraint programming [Prud'homme et al. 2015]) and heuristics to handle complex non-linear mathematical constraints [Souza et al. 2011].

Recently, Z3 [De Moura and Bjørner 2008] has emerged as leading solution for SMT solving. Developed at Microsoft Research, Z3 offers cutting-edge performance and supports a large number of theories, including bit-vectors, arrays, quantifiers, uninterpreted functions, linear integer and real arithmetic, and non-linear arithmetic. Its Z3-str [Zheng et al. 2013] extension makes it possible to treat also strings as a primitive type, allowing the solver to reason on common string operations such as concatenation, substring, and replacement. Z3 is employed in most recently appeared symbolic executors such as MAYHEM [Cha et al. 2012], SAGE [Godefroid et al. 2012], and ANGR [Shoshitaishvili et al. 2016]. Due to the extensive number of supported theories in Z3, such executors typically do not to employ additional decision procedures.

However, despite the significant advances observed over the past few years – which also made symbolic execution practical in the first place [Cadar and Sen 2013] – constraint solving remains one of the main obstacles to the scalability of symbolic execution engines, and also hinders its feasibility in the face of constraints that involve expensive theories (e.g., non-linear arithmetic) or opaque library calls.

In the remainder of this section, we address different techniques to extend the range of programs amenable to symbolic execution and to optimize the performance of constraint solving. Prominent approaches consist in: (i) reducing the size and complexity of the constraints to check, (ii) unburdening the solver by, e.g., resorting to constraint solution caching, deferring of solver queries, or concretization, and (iii) augmenting symbolic execution to handle constraints problematic for decision procedures.

[2]We refer the interested reader to [Barrett et al. 2014] for an exhaustive introduction to SMT solving, and to [Abraham et al. 2016] for a discussion of its distinctive strengths.

Constraint Reduction. A common optimization approach followed by both solvers and symbolic executors is to reduce constraints into simpler forms. For example, the *expression rewriting* optimization can apply classical techniques from optimizing compilers such as constant folding, strength reduction, and simplification of linear expressions (see, e.g., KLEE [Cadar et al. 2008]).

EXE [Cadar et al. 2006] introduces a *constraint independence* optimization that exploits the fact that a set of constraints can frequently be divided into multiple independent subsets of constraints. This optimization interacts well with query result caching strategies, and offers an additional advantage when an engine asks the solver about the satisfiability of a specific constraint, as it removes irrelevant constraints from the query. In fact, independent branches, which tend to be frequent in real programs, could lead to unnecessary constraints that would get quickly accumulated.

Another fact that can be exploited by reduction techniques is that the natural structure of programs can lead to the introduction of more specific constraints for some variables as the execution proceeds. Since path conditions are generated by conjoining new terms to an existing sequence, it might become possible to rewrite and optimize existing constraints. For instance, adding an equality constraint of the form $x := 5$ enables not only the simplification to true of other constraints over the value of the variable (e.g., $x > 0$), but also the substitution of the symbol x with the associated concrete value in the other subsequent constraints involving it. The latter optimization is also known as *implied value concretization* and, for instance, it is employed by KLEE [Cadar et al. 2008].

In a similar spirit, S^{2}E [Chipounov et al. 2012] introduces a bitfield-theory expression simplifier to replace with concrete values parts of a symbolic variable that bit operations mask away. For instance, for any 8-bit symbolic value v, the most significant bit in the value of expression $v \,|\, 10000000_2$ is always 1. The simplifier can propagate information across the tree representation of an expression, and if each bit in its value can be determined, the expression is replaced with the corresponding constant.

Reuse of Constraint Solutions. The idea of reusing previously computed results to speed up constraint solving can be particularly effective in the setting of a symbolic executor, especially when combined with other techniques such as constraint independence optimization. Most reuse approaches for constraint solving are currently based on semantic or syntactic equivalence of the constraints.

EXE [Cadar et al. 2006] caches the results of constraint solutions and satisfiability queries in order to reduce as much as possible the need for calling the solver. A cache is handled by a server process that can receive queries from multiple parallel instances of the execution engine, each exploring a different program state.

KLEE [Cadar et al. 2008] implements an incremental optimization strategy called *counterexample caching*. Using a cache, constraint sets are mapped to concrete variable assignments, or to a special null value when a constraint set is unsatisfiable. When an unsatisfiable set in the cache is a subset for a given constraint set S, S is deemed unsatisfiable as well. Conversely, when the cache contains a solution for a superset of S, the solution trivially satisfies S too. Finally, when the cache contains a solution for one or more subsets of S, the algorithm tries substituting in all the solutions to check whether a satisfying solution for S can be found.

Memoized symbolic execution [Yang et al. 2012] is motivated by the observation that symbolic execution often results in re-running largely similar sub-problems, e.g., finding a bug, fixing it, and then testing the program again to check if the fix was effective. The taken choices during path exploration are compactly encoded in a prefix tree, opening up the possibility to reuse previously computed results in successive runs.

The Green framework [Visser et al. 2012] explores constraint solution reuse across runs of not only the same program, but also similar programs, different programs, and

```
1. void test(int x, int y) {           4. int non_linear(int v) {
2.    if (non_linear(y) == x)          5.    return (v*v) % 50;
3.      if (x > y + 10) ERROR; }       6. }
```

Fig. 11: Example with non-linear constraints.

different analyses. Constraints are distilled into their essential parts through a *slicing* transformation and represented in a canonical form to achieve good reuse, even within a single analysis run. [Jia et al. 2015] presents an extension to the framework that exploits logical implication relations between constraints to support constraint reuse and faster execution times.

Lazy Constraints. [Ramos and Engler 2015] adopts a timeout approach for constraint solver queries. In their initial experiments, the authors traced most timeouts to symbolic division and remainder operations, with the worst cases occurring when an unsigned remainder operation had a symbolic value in the denominator. They thus implemented a solution that works as follow: when the executor encounters a branch statement involving an expensive symbolic operation, it will take both the true and false branches and add a *lazy* constraint on the result of the expensive operation to the path conditions. When the exploration reaches a state that satisfies some goal (e.g., an error is found), the algorithm will check for the feasibility of the path, and suppress it if deemed unreachable in a real execution.

Compared to the *eager* approach of checking the feasibility of a branch as encountered (Section 5.1), a lazy strategy may lead to a larger number of active states, and in turn to more solver queries. However, the authors report that the delayed queries are in many cases more efficient than their eager counterparts: the path constraints added after a lazy constraint can in fact narrow down the solution space for the solver.

Concretization. [Cadar and Sen 2013] discusses limitations of classical symbolic execution in the presence of formulas that constraint solvers cannot solve, at least not efficiently. A concolic executor generates some random input for the program and executes it both concretely and symbolically: a possible value from the concrete execution can be used for a symbolic operand involved in a formula that is inherently hard for the solver, albeit at the cost of possibly sacrificing soundness in the exploration.

Example. In the code fragment of Figure 11, the engine stores a non-linear constraint of the form $\alpha_x = (\alpha_y * \alpha_y) \,\%\, 50$ for the $true$ branch at line 2. A solver that does not support non-linear arithmetic fails to generate any input for the program. However, a concolic engine can exploit concrete values to help the solver. For instance, if $x = 3$ and $y = 5$ are randomly chosen as initial input parameters, then the concrete execution does not take any of the two branches. Nonetheless, the engine can reuse the concrete value of y, simplifying the previous query as $\alpha_x = 25$ due to $\alpha_y = 5$. The straightforward solution to this query can now be used by the engine to explore both branches. Notice that if the value of y is fixed to 5, then there is no way of generating a new input that takes the first but not the second branch, inducing a false negative. In this case, a trivial solution could be to rerun the program choosing a different value for y (e.g., if $y = 2$ then $x = 4$, which satisfies the first but not the second branch).

To partially overcome the incompleteness due to concretization, [Pasareanu et al. 2011] suggests *mixed concrete-symbolic solving*, which considers *all* the path constraints collectable over a path before binding one or more symbols to specific concrete values. Indeed, DART [Godefroid et al. 2005] concretizes symbols based on the path constraints collected up to a target branch. In this manner, a constraint contained in a subsequent branch in the same path is not considered and it may be not satisfiable due to already concretized symbols. If this happen, DART restarts the execution with different random concrete values, hoping to be able to satisfy the subsequent branch. The approach presented in [Pasareanu et al. 2011] requires instead to detect *solvable* constraints along a full path and to delay concretization as much as possible.

Handling Problematic Constraints. Strong SMT solvers allow executors to handle more path constraints directly, reducing the need to resort to concretization. This also results in a lower risk to incur a *blind commitment* to concrete values [Dinges and Agha 2014a], which happens when the under-approximation of path conditions from a random choice of concrete values for some variables results in an arbitrary restriction of the search space. However, the decision problem for certain classes of constraints is well known to be undecidable, e.g., like non-linear integer arithmetic, or the theory of reals with trigonometric functions often used to model real-world systems.

[Dinges and Agha 2014a] proposes a *concolic walk* algorithm that can tackle control-flow dependencies involving non-linear arithmetic and library calls. The algorithm treats assignments of values to variables as a valuation space: the solutions of the linear constraints define a polytope that can be walked heuristically, while the remaining constraints are assigned with a fitness function measuring how close a valuation point is to matching the constraint. An adaptive search is performed on the polytope as points are picked on it and non-linear constraints evaluated on them. Compared to mixed concrete-symbolic solving [Pasareanu et al. 2011], both techniques seek to avoid blind commitment. However, concolic walk does not rely on the solver for obtaining all the concrete inputs needed to evaluate complex constraints, and implements search heuristics that guide the walk on the polytope toward promising regions.

[Dinges and Agha 2014b] describes *symcretic* execution, a novel combination of symbolic backward execution (SBE) (Section 2) and forward symbolic execution. The main idea is to divide exploration into two phases. In the first phase, SBE is performed from a target point and a trace is collected for each followed path. If any problematic constraints are met during the backward exploration, the engine marks them as *potentially* satisfiable by adding a special event to the trace and continues its reversed traversal. Whenever an entry point of the program is reached along any of the followed paths, the second phase starts. The engine concretely evaluates the collected trace, trying to satisfy any constraint marked as problematic during the first phase. This is done using a heuristic search, such as the concolic walk described above. An advantage of symcretic over classical concolic execution is that it can prevent the exploration of some unfeasible paths. For instance, the backward phase may determine that a statement is guarded by an unsatisfiable branch regardless of how the statement is reached, while a traditional concolic executor would detect the unfeasibility on a per-path basis only when the statement is reached, which is unfavorable for statements "deep" in a path.

7. FURTHER DIRECTIONS

In this section we discuss how recent advances in related research areas could be applied or provide potential directions to enhance the state of the art of symbolic execution techniques. In particular, we discuss separation logic for data structures, techniques from the program verification and program analysis domains for dealing with path explosion, and symbolic computation for dealing with non-linear constraints.

7.1. Separation Logic

Checking memory safety properties for pointer programs is a major challenge in program verification. Recent years have witnessed *separation logic* (SL) [Reynolds 2002] emerging as one leading approach to reason about heap manipulations in imperative programs. SL extends Hoare logic to facilitate reasoning about programs that manipulate pointer data structures, and allows expressing complex invariants of heap configurations in a succinct manner.

At its core, a *separating conjunction* binary operator $*$ is used to assert that the heap can be partitioned in two components where its arguments separately hold. For instance, predicate $A * x \mapsto [n : y]$ says that there is a single heap cell x pointing to a record that holds y in its n field, while A holds for the rest of the heap.

Program state is modeled as a *symbolic heap* $\Pi \,\vdots\, \Sigma$: Π is a finite set of pure predicates related to variables, while Σ is a finite set of heap predicates. Symbolic heaps are SL formulas that are symbolically executed according to the program's code using an abstract semantics. SL rules are typically employed to support entailment of symbolic heaps, to infer which heap portions are not affected by a statement, and to ensure termination of symbolic execution via abstraction (e.g., using a widening operator).

A key to the success of SL lies in the local form of reasoning enabled by its $*$ operator, as it allows specifications that speak about the sole memory accessed by the code. This also fits together with the goal of deriving inductive definitions to describe mutable data structures. When compared to other verification approaches, the annotation burden on the user is rather little or often absent. For instance, the shape analysis presented in [Calcagno et al. 2011] uses bi-abduction to automatically discover invariants on data structures and compute composable procedure summaries in SL.

Several tools based on SL are available to date, for instance, for automatic memory bug discovery in user and system code, and verification of annotated programs against memory safety properties or design patterns. While some of them implement tailor-made decision procedures, [Botinčan et al. 2009; Piskac et al. 2013] have shown that provers for decidable SL fragments can be integrated in an SMT solver, allowing for complete combinations with other theories relevant to program verification. This can pave the way to applications of SL in a broader setting: for instance, a symbolic executor could use it to reason inductively about code that manipulates structures such as lists and trees. While symbolic execution is at the core of SL, to the best of our knowledge there have not been uses of SL in symbolic executors to date.

7.2. Invariants

Invariants are crucial for verifiers that can prove programs correct against their full functional specification. An invariant is a predicate true for an initial state and for each state reachable from it. Invariant synthesis normally targets inductive predicates, which are closed under the state transition relation (i.e., they make no reference to past behavior). All inductive predicates are invariants, but the converse is not true.

Leveraging invariants can be beneficial to symbolic executors, in order to compactly capture the effects of a loop and reason about them. Unfortunately, we are not aware of symbolic executors taking advantage of this approach. One of the reasons might lie in the difficulty of computing loop invariants without requiring manual intervention from domain experts. In fact, lessons from the verification practice suggest that providing loop invariants is much harder compared to other specification elements such as method pre/post-conditions.

However, many researchers have recently explored techniques for inferring loop invariants automatically or with little human help [Furia et al. 2014], which might be of interest for the symbolic execution community for a more efficient handling of loops.

Termination analysis has been applied to verify program termination for industrial code: a formal argument is typically built by using one or more ranking functions over all the possible states in the program such that for every state transition, at least one function decreases [Cook et al. 2006]. Ranking functions can be constructed in a number of ways, e.g., by lazily building an invariant using counterexamples from traversed loop paths [Gonnord et al. 2015]. A termination argument can also be built by reasoning over transformed programs where loops are replaced with summaries based on transition invariants [Tsitovich et al. 2011]. It has been observed that most loops in practice have relatively simple termination arguments [Tsitovich et al. 2011]: the discovered invariants may thus not be rich enough for a verification setting [Galeotti et al. 2015]. However, a constant or parametric bound on the number of iterations may still be computed from a ranking function and an invariant [Gonnord et al. 2015].

Predicate abstraction is a form of abstract interpretation over a domain constructed using a given set of predicates, and has been used to infer universally quantified loop invariants [Flanagan and Qadeer 2002], which are useful when manipulating arrays. Predicates can be heuristically collected from the code or supplied by the user: it would be interesting to explore a mutual reinforcing combination with symbolic execution, with additional useful predicates being originated during the symbolic exploration.

LoopFrog [Kroening et al. 2008] replaces loops using a symbolic abstract transformer with respect to a set of abstract domains, obtaining a conservative abstraction of the original code. Abstract transformers are computed starting from the innermost loop, and the output is a loop-free summary of the program that can be handed to a model checker for verification. This approach can also be applied to non-recursive function calls, and might deserve some investigation in symbolic executors.

Loop invariants can also be extracted using *interpolation*, a general technique that has already been applied in symbolic execution for different goals (Section 5.3).

7.3. Function Summaries

Function summaries (Section 5.2) have largely been employed in static and dynamic program analysis, especially in program verification. A number of such works could offer interesting opportunities to advance the state of the art in symbolic execution. For instance, the Calysto static checker [Babic and Hu 2008] walks the call graph of a program to construct a symbolic representation of the effects of each function, i.e., return values, writes to global variables, and memory locations accessed depending on its arguments. Each function is processed once, possibly inlining effects of small ones at their call sites. Static checkers such as Calysto and Saturn [Xie and Aiken 2005] trade scalability for soundness in summary construction, as they unroll loops only to a small number of iterations: their use in a symbolic execution setting may thus result in a loss of soundness. More fine-grained summaries are constructed in [Engler and Ashcraft 2003] by taking into account different input conditions using a summary cache for memoizing the effects of a function.

[Sery et al. 2012b] proposes a technique to extract function summaries for model checking where multiple specifications are typically checked one a time, so that summaries can be reused across verification runs. In particular, they are computed as over-approximations using interpolation (Section 5.3) and refined across runs when too weak. The strength of this technique lies in the fact that an interpolant-based summary can capture all the possible execution traces through a function in a more compact way than the function itself. The technique has later been extended to deal with nested function calls in [Sery et al. 2012a].

7.4. Program Analysis and Optimization

We believe that the symbolic execution practice might further benefit from solutions that have been proposed for related problems in the programming languages realm. For instance, in the parallel computing community transformations such as *loop coalescing* [Bacon et al. 1994] can restructure nested loops into a single loop by flattening the iteration space of their indexes. Such a transformation could potentially simplify a symbolic exploration, empowering search heuristics and state merging strategies.

Loop unfolding [Song and Kavi 2004] may possibly be interesting as well, as it allows exposing “well-structured” loops (e.g., showing invariant code, or having constants or affine functions as subscripts of array references) by peeling several iterations.

Program synthesis automatically constructs a program satisfying a high-level specification [Pnueli and Rosner 1989]. The technique has caught the attention of the verification community since [Solar Lezama 2008] has shown how to find programs as a solution to SAT problems. In Section 4 we discussed its usage in [Jeon et al. 2016] to produce compact models for complex Java frameworks: the technique takes as inputs classes, methods and types from a framework, along with tutorial programs (typically those provided by the vendor) that exercise its parts. We believe this approach deserves further investigation in the context of the path explosion problem. It could potentially be applied to software modules such as standard libraries to produce concise models that allow for a more scalable exploration of the search space, as synthesis can capture an external behavior while abstracting away entanglements of the implementation.

7.5. Symbolic Computation

Although the satisfiability problem is known to be NP-hard already for SAT, the mathematical developments over the past decades have produced several practically applicable methods to solve arithmetic formulas. In particular, advances in *symbolic computation* have produced powerful methods such as Gröbner bases for solving systems of polynomial constraints, cylindrical algebraic decomposition for real algebraic geometry, and virtual substitution for non-linear real arithmetic formulas [Abraham 2015].

While SMT solvers are very efficient at combining theories and heuristics when processing complex expressions, they make use of symbolic computation techniques only to a little extent, and their support for non-linear real and integer arithmetic is still in its infancy [Abraham 2015]. To the best of our knowledge, only Z3 [De Moura and Bjørner 2008] and SMT-RAT [Corzilius et al. 2015] can reason about them both.

[Abraham 2015] states that using symbolic computation techniques as theory plugins for SMT solvers is a promising symbiosis, as they provide powerful procedures for solving conjunctions of arithmetic constraints. The realization of this idea is hindered by the fact that available implementations of such procedures do not comply with the incremental, backtracking and explanation of inconsistencies properties expected of SMT-compliant theory solvers. One interesting project to look at is SC^2 [Abraham et al. 2016], whose goal is to create a new community aiming at bridging the gap between symbolic computation and satisfiability checking, combining the strengths of both worlds in order to pursue problems currently beyond their individual reach.

Further opportunities to increase efficiency when tackling non-linear expressions might be found in the recent advances in *symbolic-numeric computation* [Grabmeier et al. 2003]. In particular, these techniques aim at developing efficient polynomial solvers by combining numerical algorithms, which are very efficient in approximating local solutions but lack a global view, with the guarantees from symbolic computation techniques. This hybrid techniques can extend the domain of efficiently solvable problems, and thus be of interest for non-linear constraints from symbolic execution.

8. CONCLUSIONS

Symbolic execution techniques have evolved significantly in the last decade, with notable applications to compelling problems from several domains like software testing (e.g., test input generation, regression testing), security (e.g., exploit generation, authentication bypass), and code analysis (e.g., program deobfuscation, dynamic software updating). This trend has not only improved existing solutions, but also led to novel ideas and, in some cases, to major practical breakthroughs. For instance, the push for scalable automated program analyses in security has culminated in the 2016 DARPA Cyber Grand Challenge, which hosted systems for detecting and fixing vulnerabilities in unknown software with no human intervention, such as ANGR [Shoshitaishvili et al. 2016] and MAYHEM [Cha et al. 2012], that competed for nearly $4M in prize money.

This survey has discussed some of the key aspects and challenges of symbolic execution, presenting for a broad audience the basic design principles of symbolic executors and the main optimization techniques. We hope it will help non-experts grasp the key inventions in this exciting line of research, inspiring further work and new ideas.

ELECTRONIC APPENDIX

The online appendix of this manuscript discusses a selection of prominent applications of symbolic execution techniques, addresses further challenges that arise in the analysis of programs in binary form, and provides a list of popular symbolic engines.

REFERENCES

Erika Abraham. 2015. Building Bridges Between Symbolic Computation and Satisfiability Checking. In *Proc. 2015 ACM on Int. Symp. on Symbolic and Algebraic Computation (ISSAC'15)*. ACM, 1–6.

Erika Abraham, John Abbott, Bernd Becker, Anna M. Bigatti, Martin Brain, Bruno Buchberger, Alessandro Cimatti, James H. Davenport, Matthew England, Pascal Fontaine, Stephen Forrest, Alberto Griggio, Daniel Kroening, Werner M. Seiler, and Thomas Sturm. 2016. SC2: Satisfiability Checking Meets Symbolic Computation. In *Proc. 9th Int. Conf. on Intelligent Computer Math. (CICM'16)*. Springer, 28–43.

Saswat Anand. 2012. *Techniques to Facilitate Symbolic Execution of Real-world Programs*. Ph.D. Dissertation. Atlanta, GA, USA. AAI3531671.

Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-driven Compositional Symbolic Execution. In *Proc. Theory and Practice of Software, 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. 367–381.

Saswat Anand, Alessandro Orso, and Mary Jean Harrold. 2007. Type-dependence Analysis and Program Transformation for Symbolic Execution. In *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*. 117–133.

Saswat Anand, Corina S. Pasareanu, and Willem Visser. 2009. Symbolic Execution with Abstraction. *Int. J. Software Tools Technol. Transf.* 11, 1 (2009), 53–67.

Thanassis Avgerinos. 2014. *Exploiting Trade-offs in Symbolic Execution for Identifying Security Bugs*. Ph.D. Dissertation. http://repository.cmu.edu/cgi/viewcontent.cgi?article=1478&context=dissertations.

Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In *Proc. Network and Distributed System Security Symp. (NDSS'11)*.

Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proc. 36th Int. Conf. on Software Engineering (ICSE'14)*. ACM, 1083–1094.

Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and Precise Extended Static Checking. In *Proc. 30th Int. Conf. on Software Engineering (ICSE'08)*. ACM, 211–220.

David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-performance Computing. *ACM Computing Surveys (CSUR)* 26, 4 (1994), 345–420.

Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. 2006. Thorough Static Analysis of Device Drivers. In *Proc. 1st ACM SIGOPS/EuroSys European Conf. on Comp. Systems (EuroSys'06)*. ACM, 73–85.

Clark Barrett, Daniel Kroening, and Thomas Melham. 2014. *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories*. London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering.

Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proc. USENIX Annual Technical Conf. (ATEC'05)*. USENIX Association, 41–41.

Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Proc. 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. 351–366.

Matko Botinčan, Matthew Parkinson, and Wolfram Schulte. 2009. Separation Logic Verification of C Programs with an SMT Solver. *Electronic Notes in Theoretical Comp. Science* 254 (2009), 5–23.

Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proc. of Int. Conf. on Reliable Software*. ACM, 234–245.

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proc. 23rd Int. Conf. on Computer Aided Verification (CAV'11)*. 463–469.

Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-world Software Testing. In *Proc. 6th Conf. on Comp. Systems (EuroSys'11)*. 183–198.

Cristian Cadar. 2015. Targeted Program Transformations for Symbolic Execution. In *Proc. 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, 906–909.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proc. 8th USENIX Conf. on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 209–224.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proc. 13th ACM Conf. on Computer and Communications Security (CCS'06)*. ACM, 322–335.

Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (2011).

Matteo Ceccarello and Oksana Tkachuk. 2014. Automated Generation of Model Classes for Java PathFinder. *SIGSOFT Software Engineering Notes* 39, 1 (2014), 1–5.

Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proc. 2012 IEEE Symp. on Sec. and Privacy (SP'12)*. IEEE Comp. Society, 380–394.

Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proc. 30th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'09)*. ACM, 363–374.

Ting Chen, Xiaodong Lin, Jin Huang, Abel Bacchus, and Xiaosong Zhang. 2015. An Empirical Investigation into Path Divergences for Concolic Execution Using CREST. *Security and Communication Networks* 8, 18 (2015), 3667–3681.

Ting Chen, Xiao-Song Zhang, Shi-Ze Guo, Hong-Yuan Li, and Yue Wu. 2013. State of the Art: Dynamic Symbolic Execution for Automated Test Generation. *Future Gen. Comput. Syst.* 29, 7 (2013), 1758–1773.

Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 2:1–2:49.

Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic Crosschecking of Floating-point and SIMD Code. In *Proc. Sixth Conf. on Computer Systems (EuroSys'11)*. ACM, 315–328.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination Proofs for Systems Code. In *Proc. 27th ACM SIGPLAN Conf. on Programming Language Design and Implementation*. 415–426.

Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Rethinking Pointer Reasoning in Symbolic Execution. In *Proc. 32nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'17)*. 613–618.

Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Abraham. 2015. SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving. In *Proc. 18th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'15)*, Marijn Heule and Sean Weaver (Eds.). 360–368.

William Craig. 1957. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. Symbolic Logic* 22, 3 (1957), 269–285.

Christoph Csallner and Yannis Smaragdakis. 2005. Check 'N' Crash: Combining Static Checking and Testing. In *Proc. 27th Int. Conf. on Software Engineering (ICSE'05)*. ACM, 422–431.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. Theory and Practice of Software, 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. 337–340.

Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (2011), 69–77.

Xianghua Deng, Jooyong Lee, and Robby. 2012. Efficient and Formal Generalized Symbolic Execution. *Automated Software Engineering* 19, 3 (2012), 233–301.

Peter Dinges and Gul Agha. 2014a. Solving Complex Path Conditions Through Heuristic Search on Induced Polytopes. In *Proc. 22nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*. 425–436.

Peter Dinges and Gul Agha. 2014b. Targeted Test Input Generation Using Symbolic-concrete Backward Execution. In *Proc. 29th ACM/IEEE Int. Conf. on Automated Software Engineering (ASE'14)*. 31–36.

Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. 2015. Studying the Influence of Standard Compiler Optimizations on Symbolic Execution. In *Proc. 2015 IEEE 26th Int. Symp. on Software Reliability Engineering*. 205–215.

Evelyn Duesterwald (Ed.). 2004. *Analyzing Memory Accesses in x86 Executables*. Springer.

Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. 2009. Precise Pointer Reasoning for Dynamic Test Generation. In *Proc. 18th Int. Symp. on Software Testing and Analysis (ISSTA'09)*. ACM, 129–140.

Dawson R. Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proc,. 19th ACM Symp. on Operating Systems Principles (SOSP'03)*. ACM, 237–252.

Dawson R. Engler and Daniel Dunbar. 2007. Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable. In *Proc. of 2007 Int. Symp. on Soft. Test. and Analysis (ISSTA'07)*. 1–4.

Cormac Flanagan and Shaz Qadeer. 2002. Predicate Abstraction for Software Verification. In *Proc. of 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'02)*. ACM, 191–202.

Carlo A. Furia, Bertrand Meyer, and Sergey Velder. 2014. Loop Invariants: Analysis, Classification, and Examples. *ACM Computing Surveys (CSUR)* 46, 3, Article 34 (2014).

Juan P. Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, and Andreas Zeller. 2015. Inferring Loop Invariants by Mutation, Dynamic Analysis, and Static Checking. *IEEE Transactions on Software Engineering (TSE)* 41, 10 (2015), 1019–1037.

Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays. In *Proc. 19th Int. Conf. on Computer Aided Verification (CAV'07)*. 519–531.

Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *Proc. 34th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'07)*. 47–54.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'05)*. 213–223.

Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. Network and Distributed System Security Symp. (NDSS'08)*.

Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (2012), 20:20–20:27 pages.

Patrice Godefroid and Daniel Luchaup. 2011. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proc. 2011 Int. Symp. on Software Testing and Analysis (ISSTA'11)*. ACM, 23–33.

Laure Gonnord, David Monniaux, and Gabriel Radanne. 2015. Synthesis of Ranking Functions Using Extremal Counterexamples. In *Proc. 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'15)*. ACM, 608–618.

Johannes Grabmeier, Erich Kaltofen, and Volker Weispfenning. 2003. *Computer Algebra Handbook: Foundations, Applications, Systems*. Vol. 1. Springer Science & Business Media, 109–124.

Trevor Hansen, Peter Schachte, and Harald Søndergaard. 2009. Runtime Verification. Chapter State Joining and Splitting for the Symbolic Execution of Binaries, 76–92.

William E. Howden. 1977. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering (TSE)* 3, 4 (1977), 266–278.

Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. 2013. Boosting Concolic Testing via Interpolation. In *Proc. 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, 48–58.

Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. 2012a. TRACER: A Symbolic Execution Tool for Verification. In *Proc. 24th Int. Conf. on Comp. Aided Verification (CAV'12)*. 758–766.

Joxan Jaffar, Jorge A. Navas, and Andrew E. Santosa. 2012b. Unbounded Symbolic Execution for Program Verification. In *Proc. 2nd Int. Conf. on Runtime Verification (RV'11)*. 396–411.

Joxan Jaffar, Andrew E. Santosa, and Răzvan Voicu. 2009. An Interpolation Method for CLP Traversal. In *Proc. 15th Int. Conf. on Principles and Practice of Constraint Programming (CP'09)*. 454–469.

Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. 2016. Synthesizing Framework Models for Symbolic Execution. In *Proc. 38th Int. Conf. on Software Engineering (ICSE'16)*. ACM, 156–167.

Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *Proc. 2015 Int. Symp. on Software Testing and Analysis (ISSTA'15)*. 177–187.

Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. 2010. Mixing Type Checking and Symbolic Execution. In *Proc. 31st ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI'10)*. 436–447.

Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Proc. 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*. Springer-Verlag, 553–568.

James C. King. 1975. A New Approach to Program Testing. In *Proc. Int. Conf. on Reliable Software*. ACM, 228–233.

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2008. Loop Summarization Using Abstract Transformers. In *Proc. 6th Int. Symp. on Automated Technology for Verification and Analysis (ATVA'08)*. 111–125.

Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In *Proc. 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'12)*. ACM, 193–204.

You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *Proc. ACM SIGPLAN Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13)*. 19–32.

Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Proc. 18th Int. Conf. on Static Analysis (SAS'11)*. 95–111.

Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *Proc. 29th Int. Conf. on Software Engineering (ICSE'07)*. IEEE Computer Society, 416–426.

Rupak Majumdar and Ru-Gang Xu. 2009. Reducing Test Inputs Using Information Partitions. In *Proc. 21st Int. Conf. on Computer Aided Verification (CAV'09)*. Springer-Verlag, Berlin, Heidelberg, 555–569.

Kenneth L. McMillan. 2010. Lazy Annotation for Program Testing and Verification. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV'10)*. 104–118.

Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification & Reliability* 14, 2 (2004), 105–156.

Corina S. Pasareanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proc. IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'10)*. ACM, 179–180.

Corina S. Pasareanu, Neha Rungta, and Willem Visser. 2011. Symbolic Execution with Mixed Concrete-symbolic Solving. In *Proc. 2011 Int. Symp. on Software Testing and Analysis (ISSTA'11)*. ACM, 34–44.

Corina S. Pasareanu and Willem Visser. 2009. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *Int. Journal on Software Tools for Technology Transfer* 11, 4 (2009), 339–353.

David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating Array Constraints in Symbolic Execution. In *Proc. 26th ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA'17)*. ACM, 68–78.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *Proc. 25th Int. Conf. on Computer Aided Verification (CAV'13)*. 773–789.

Amir Pnueli and Roni Rosner. 1989. On the Synthesis of a Reactive Module. In *Proc. 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'89)*. ACM, 179–190.

Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. 2015. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.

Dawei Qi, Hoang D. T. Nguyen, and Abhik Roychoudhury. 2013. Path Exploration Based on Symbolic Output. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 4, Article 32 (2013).

David A. Ramos and Dawson R. Engler. 2015. Under-constrained Symbolic Execution: Correctness Checking for Real Code. In *Proc. 24th USENIX Conf. on Security Symp. (SEC'15)*. USENIX Association, 49–64.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. 17th Annual IEEE Symp. on Logic in Computer Science (LICS'02)*. IEEE Computer Society, 55–74.

Nicolas Rosner, Jaco Geldenhuys, Nazareno M. Aguirre, Willem Visser, and Marcelo F. Frias. 2015. BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support. *IEEE Transactions on Software Engineering (TSE)* 41, 7 (2015), 639–660.

Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended Symbolic Execution on Binary Programs. In *Proc. 18th Int. Symp. on Software Testing and Analysis*. 225–236.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proc. 2010 IEEE Symp. on Security and Privacy (SP'10)*. IEEE Computer Society, 317–331.

Daniel Schwartz-Narbonne, Martin Schaf, Dejan Jovanovic, Philipp Rümmer, and Thomas Wies. 2015. Conflict-Directed Graph Coverage. In *NASA Formal Methods: 7th Int. Symp.* 327–342.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proc. 10th European Software Engineering Conf. Held Jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, 263–272.

Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2012a. Incremental Upgrade Checking by Means of Interpolation-based Function Summaries. In *2012 Formal Methods in Computer-Aided Design (FMCAD'12)*. 114–121.

Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2012b. Interpolation-Based Function Summaries in Bounded Model Checking. In *Proc. 7th Int. Haifa Verification Conf. on Hardware and Software: Verification and Testing (HVC'11)*. 160–175.

Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *22nd Annual Network and Distributed System Security Symp. (NDSS'15)*.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symp. on Security and Privacy (SP'16)*. 138–157.

Jiri Slaby, Jan Strejcek, and Marek Trtik. 2013. Compact Symbolic Execution. In *11th Int. Symp. on Automated Technology for Verification and Analysis (ATVA'13)*. 193–207.

Armando Solar Lezama. 2008. *Program Synthesis By Sketching*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proc. 4th Int. Conf. on Information Systems Security ((ICISS'08)*. 1–25.

Litong Song and Krishna Kavi. 2004. What Can We Gain by Unfolding Loops? *SIGPLAN Not.* 39, 2 (2004), 26–33.

Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Pasareanu. 2011. CORAL: Solving Complex Constraints for Symbolic Pathfinder. In *Proc. 3rd Int. NASA Formal Methods Symp.* 359–374.

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23nd Annual Network and Distr. System Sec. Symp. (NDSS'16)*.

Aditya Thakur, Junghee Lim, Akash Lal, Amanda Burton, Evan Driscoll, Matt Elder, Tycho Andersen, and Thomas Reps. 2010. Directed Proof Generation for Machine Code. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV'10)*. Springer-Verlag, 288–305.

Marek Trtik and Jan Strejček. 2014. *Symbolic Memory with Pointers*. Springer Int. Publishing, 380–395.

Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. 2011. Loop Summarization and Termination Analysis. In *Proc. Theory and Practice of Software, Proc. 17th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11/ETAPS'11)*. 81–95.

Heila van der Merwe, Oksana Tkachuk, Brink van der Merwe, and Willem Visser. 2015. Generation of Library Models for Verification of Android Applications. *SIGSOFT Software Engineering Notes* 40, 1 (2015), 1–5.

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proc. ACM SIGSOFT 20th Int. Symp. on the Foundations of Software Engineering (FSE'12)*. ACM, Article 58.

Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT Int. Symp. on Software Testing and Analysis*. ACM, 97–107.

Jonas Wagner, Volodymyr Kuznetsov, and George Candea. 2013. Overify: Optimizing Programs for Fast Verification. In *Proc. 14th USENIX Conf. on Hot Topics in Operating Systems*. USENIX Association.

Haijun Wang, Ting Liu, Xiaohong Guan, Chao Shen, Qinghua Zheng, and Zijiang Yang. 2017. Dependence Guided Symbolic Execution. *IEEE Transactions Software Engineering (TSE)* 43, 3 (2017), 252–271.

Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357.

Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. Precise Identification of Problems for Structural Test Generation. In *Proc. 33rd Int. Conf. on Software Engineering (ICSE'11)*. 611–620.

Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. 2009 IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN'09)*. 359–368.

Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis. In *Proc. 2016 24th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE'16)*. 61–72.

Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In *Proc. 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'05)*. ACM, 351–363.

Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. 2012. Memoized Symbolic Execution. In *Proc. 2012 Int. Symp. on Software Testing and Analysis (ISSTA'12)*. ACM, 144–154.

Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed Incremental Symbolic Execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 1, Article 3 (2014).

Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2015. Postconditioned Symbolic Execution. In *2015 IEEE 8th Int. Conf. on Software Testing, Verification and Validation (ICST)*.

Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular Property Guided Dynamic Symbolic Execution. In *Proc. 37th Int. Conf. on Software Engineering (ICSE'15)*. 643–653.

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proc. 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 114–124.