Computing Surveys

A Survey of Symbolic Execution Techniques

Journal:	*Computing Surveys*
Manuscript ID	Draft
Paper:	Regular Paper
Date Submitted by the Author:	n/a
Complete List of Authors:	Baldoni, Roberto; Universita degli Studi di Roma La Sapienza COPPA, EMILIO; Universita degli Studi di Roma La Sapienza D'Elia, Daniele Cono; Universita degli Studi di Roma La Sapienza Demetrescu, Camil; Universita degli Studi di Roma La Sapienza Finocchi, Irene; Universita degli Studi di Roma La Sapienza
Computing Classification Systems:	Software verification, Dynamic analysis, Software testing and debugging, Software and application security

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;** *Dynamic analysis; Software testing and debugging;* •**Security and privacy** → Software and application security;

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

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.* 9, 4, Article 39 (March 2010), 35 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 in the context of software testing to check whether a certain property can be violated by a program [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.

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 underapproximate 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 [Biere et al. 2009], 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 heterogenous 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 one third of the bugs discovered during the development of Windows 7, which were missed by other program analyses and blackbox testing techniques [Godefroid et al. 2012].

In this article, we survey the main aspects of symbolic execution and discuss its extensive usage in software testing and computer security applications, where software vulnerabilities can be found by symbolically executing programs at the level of either source or binary code. 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 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:

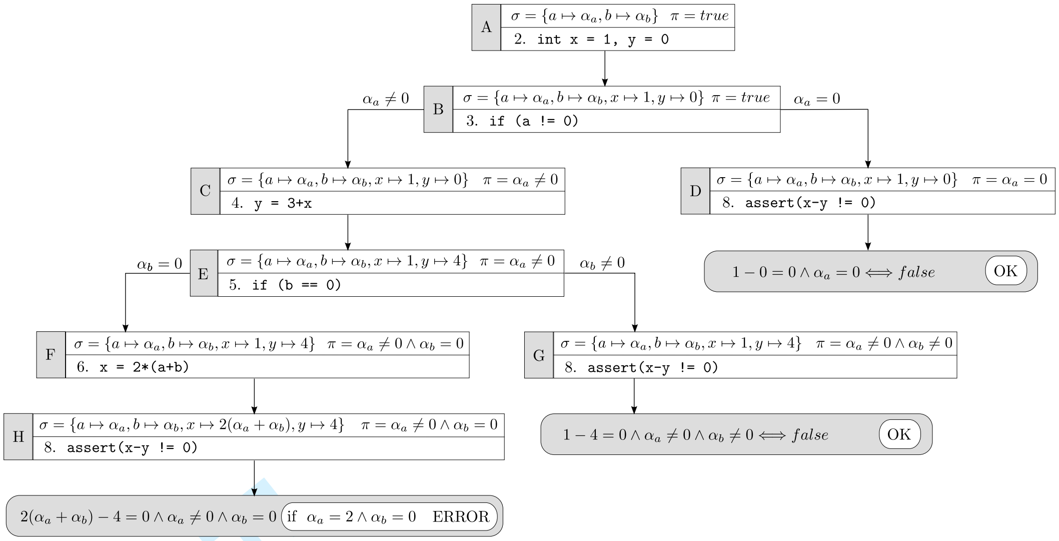

A
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\}$ $\pi = true$
2. int x = 1, y = 0
$\alpha_a \neq 0$
B
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = true$
3. if (a != 0)
$\alpha_a = 0$
C
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a \neq 0$
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
$\alpha_b = 0$
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)
$\alpha_b \neq 0$
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)
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)
$1 - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b \neq 0 \Longleftrightarrow false$
OK
$2(\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0$
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 2 and Section 5, respectively).

— 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 $\mathtt{x} + 3$, 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 a *model checker* [Biere et al. 2009] 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 as 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? Any arbitrarily complex object can be regarded as an array of bytes and each byte associated with a distinct symbol. However, when possible, exploiting structural properties of the data may be more convenient: for instance, relational bounds on the class fields in object-oriented languages could be used for refining the search performed by symbolic execution.

— *Environment*: how does the symbolic engine handle interactions with the environment? Real-world applications constantly interact with the environment (e.g., the file system or the network) through libraries and system calls. These interactions may cause side-effects (such as the creation of a file) that could later affect the execution and must be therefore taken into account. Evaluating any possible interaction outcome is generally unfeasible: it could generate a large number of execution states, of which only a small number can actually happen in a non-symbolic scenario. A typical strategy is to consider popular library and system routines and create models that can help the symbolic engine analyze only significant outcomes.

— *Loops*: how does the symbolic engine handle loops? Choosing the number of loop iterations to analyze is especially critical when this number cannot be determined in advance (e.g., depends on an input parameter). The naive approach of unrolling iterations for every valid bound would result in a prohibitively large number of states. Typical solutions are to compute an underapproximation of the analysis by limiting the number of iterations to some value k, thus trading speed for soundness. Other approaches infer loop invariants through static analysis and use them to merge equivalent states.

— *State space explosion and path selection*: 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. In practice, heuristics are used to guide exploration and prioritize certain states first (e.g., to maximize code coverage). In addition, symbolic engines can implement efficient mechanisms for evaluating multiple states in parallel without running out of resources.

— *Constraint solver*: what can a constraint solver do in practice? Constraint solvers suffer from a number of limitations. They can typically handle complex constraints in a reasonable amount of time only if they are made of linear expressions over their constituents. Symbolic execution engines normally implement a number of optimizations to make queries as much *solver-friendly* as possible, for instance by splitting queries into independent components to be processed separately or by performing algebraic simplifications.

— *Binary code*: what issues can arise when symbolically executing binary code? While the warm-up example of Section 1.1 is written in C, in several scenarios binary code is the only available representation of a program. However, having the source code of an application can make symbolic execution significantly easier, as it can exploit high-level properties (e.g., object shapes) that can be inferred statically by analyzing the source code.

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. 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 8 address the key challenges that we listed in Section 1.2. Prominent applications based on symbolic execution techniques are discussed in Section 9, while concluding remarks are addressed in Section 10.

2. SYMBOLIC EXECUTION ENGINES

In this section we describe some important principles for the design of symbolic executors as well as 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. Concrete, Symbolic, and Concolic 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, for a variety of reasons.

First, as extensively discussed in Section 6, the number of control flow paths to be generated and analyzed could be prohibitively large, due to branch instructions and loops. In the worst case, if the code contains an unbounded loop, symbolic execution could keep running forever, generating a potentially infinite number of paths (we refer to Section 5 for an example).

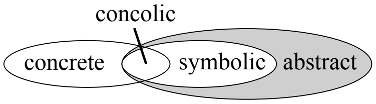


Fig. 3: Concrete and abstract execution machine models.

Moreover, as observed in Section 1, symbolic engines are clients of SMT solvers, which are continuously invoked during the analysis. Although powerful SMT solvers are currently available, the time spent in constraint solving is still one of the main performance barriers for symbolic engines. It may also happen that the program yields constraints that the solver cannot handle well (e.g., non-linear constraints), in spite of the fact that symbolic executors often use more than one solver in order to support as many decidable logical fragments as possible.

A standard approach to limit the resources (running time and space usage) required by the execution engine and to handle complex constraints 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. The basic idea is to have the concrete execution drive the symbolic execution (see also Figure 3). Besides the symbolic store and the path constraints, a concolic execution engine also 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. 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.

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 $A \rightsquigarrow B \rightsquigarrow C \rightsquigarrow E \rightsquigarrow F$.

As shown by the example, the symbolic information maintained during a concrete run can be exploited by the execution engine, for instance, to obtain new inputs and explore new control flow paths. We will further discuss this aspect in Section 6.2.

It is worth noticing that concolic execution trades soundness for performance: false negatives are indeed possible, because some program executions – and therefore possible erroneous behaviors – may be missed. In the literature, this is also regarded as an *under-approximate* form of program analysis.

Many papers exploit variants of concolic execution or different ways of mixing concrete and symbolic runs. For instance, in *execution-generated testing* (see, e.g., KLEE [Cadar et al. 2008], EXE [Cadar et al. 2006], and [Cadar and Sen 2013], the symbolic engine always executes concretely the operations that involve only concrete values. This makes it possible to reason even over complex (e.g., non-linear) operations if they involve only concrete values. *Selective symbolic execution* [Chipounov et al.

2012] takes a different approach, by interleaving portions of code that are concretely run with fully symbolic phases. The interleaving must be done carefully in order to preserve the meaningfulness of the whole exploration. When an argument x for a function call to concretize is symbolic, the engine should convert it to some concrete value in order to perform the call, which is equivalent to corseting the exploration to a single path in the callee. When the call returns and the symbolic phase resumes, the concrete value for x becomes part of the path constraints for the remainder of the exploration. However, a large number of paths may be then excluded. S^2E [Chipounov et al. 2012] presents a systematic approach to consistently cross the symbolic/concrete boundary in both directions: it describes a strategy to deal with constraints introduced on symbolic values as a consequence of concretization, and introduces a number of consistency models – where a state is *consistent* when there exists a feasible path to it from the initial state – which suit different analyses. Constraints updated to account for concrete values are marked as *soft*, and whenever a branch in the symbolic domain is disabled because of a soft constraint, execution goes back and picks a value for the concrete call that would enable that branch. Throughout this article, we will see other uses of concretization (see, e.g., Section 3 and Section 7) and of concolic execution (see Section 6).

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 make forward progress 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 tradeoffs 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 executors* – 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, since many active states need to be kept in memory, they put a huge burden on memory consumption, 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]. Moreover, 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 implemen-

Symbolic engine	**References**	**Project URL** (last retrieved: August 2016)
CUTE	[Sen et al. 2005]	–
DART	[Godefroid et al. 2005]	–
JCUTE	[Sen and Agha 2006]	https://github.com/osl/jcute
KLEE	[Cadar et al. 2006; Cadar et al. 2008]	https://klee.github.io/
SAGE	[Godefroid et al. 2008; Elkarablieh et al. 2009]	–
BITBLAZE	[Song et al. 2008]	http://bitblaze.cs.berkeley.edu/
CREST	[Burnim and Sen 2008]	https://github.com/jburnim/crest
PEX	[Tillmann and De Halleux 2008]	http://research.microsoft.com/en-us/projects/pex/
RUBYX	[Chaudhuri and Foster 2010]	–
JAVA PATHFINDER	[Păsăreanu and Rungta 2010]	http://babelfish.arc.nasa.gov/trac/jpf
OTTER	[Reisner et al. 2010]	https://bitbucket.org/khooyp/otter/
BAP	[Brumley et al. 2011]	https://github.com/BinaryAnalysisPlatform/bap
CLOUD9	[Bucur et al. 2011]	http://cloud9.epfl.ch/
MAYHEM	[Cha et al. 2012]	–
SYMDROID	[Jeon et al. 2012]	–
S^{2}E	[Chipounov et al. 2012]	http://s2e.epfl.ch/
FUZZBALL	[Martignoni et al. 2012; Caselden et al. 2013]	http://bitblaze.cs.berkeley.edu/fuzzball.html
JALANGI	[Sen et al. 2013]	https://github.com/Samsung/jalangi2
PATHGRIND	[Sharma 2014]	https://github.com/codelion/pathgrind
KITE	[do Val 2014]	http://www.cs.ubc.ca/labs/isd/Projects/Kite
SYMJS	[Li et al. 2014]	–
CIVL	[Siegel et al. 2015]	http://vsl.cis.udel.edu/civl/
KEY	[Hentschel et al. 2014]	http://www.key-project.org/
ANGR	[Shoshitaishvili et al. 2015; Shoshitaishvili et al. 2016]	http://angr.io/
TRITON	[Saudel and Salwan 2015]	http://triton.quarkslab.com/
PYEXZ3	[Ball and Daniel 2015]	https://github.com/thomasjball/PyExZ3
JDART	[Luckow et al. 2016]	https://github.com/psycopaths/jdart
CATG	–	https://github.com/ksen007/janala2
PYSYMEMU	–	https://github.com/feliam/pysymemu/
MIASM	–	https://github.com/cea-sec/miasm

Fig. 4: Selection of symbolic execution engines, along with their reference article(s) and software project web site (if any).

tation 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, the concrete state is restored and the online exploration resumes.

2.3. Caching

Caching is a powerful technique to achieve time-space tradeoffs and is embodied in symbolic executors in different ways. Most prominently:

— *Function caching.* A function f, and more in general any part of a program, may be called multiple times during an execution, either at the same calling context or at different ones. The traditional symbolic execution approach requires to symbolically execute f at each call. [Godefroid 2007] proposes a compositional approach that dynamically generates *function summaries*, allowing the symbolic executor to effectively reuse prior discovered analysis results. A similar idea has been also proposed in [Boonstoppel et al. 2008]. The main intuition is that, if two program states differ only for some program values that are not read later, the executions generated by the two program states will produce the same side effects. Side effects of a portion of code can be therefore cached and possibly reused later.

— *Loop summarization.* In order to avoid redundant executions of the same loop under the same program state, loop summaries can be computed and cached for later reuse, similarly to function summaries. We refer to Section 5 for details on a loop summarization strategy proposed in [Godefroid and Luchaup 2011].

— *Constraint reuse.* In order to speed up constraint solving, different works support the reuse of constraint solutions based on syntactic or semantic equivalence of the constraints. Examples are given in EXE [Cadar et al. 2006], KLEE [Cadar et al. 2008], and [Yang et al. 2012; Visser et al. 2012]. We will further discuss this optimization in Section 7.

2.4. Tools

Table 4 lists a number of symbolic execution engines that have worked as incubators for several of the techniques surveyed in this article. The novel contributions introduced by tools that represented milestones in the area are described in the appropriate sections throughout the article.

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 general, 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 have a significant influence on the coverage achieved by symbolic execution, as well as on 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 derived from user input instead of a concrete value. In the remainder of this section, we discuss a number of popular solutions.

3.1. Fully Symbolic Memory

At the one end of the spectrum, 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 [Trtík and Strejček 2014]). Two fundamental approaches, pioneered by King in its 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 example 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.

— *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

```
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?

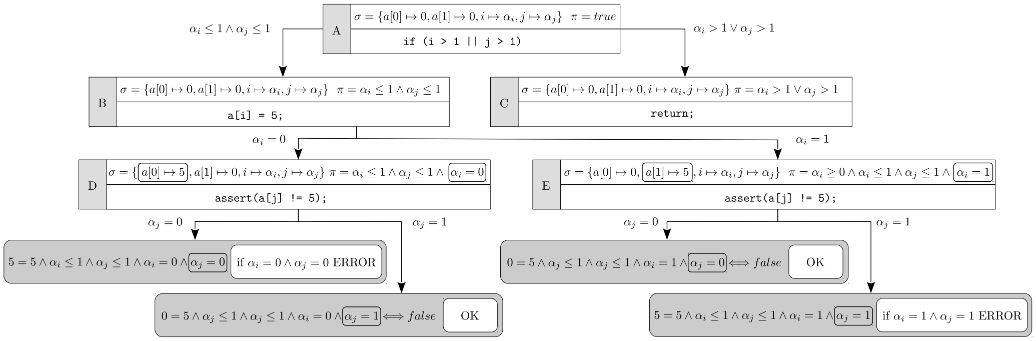

A
$\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = true$
if (i > 1 || j > 1)
$\alpha_i \leq 1 \wedge \alpha_j \leq 1$
$\alpha_i > 1 \vee \alpha_j > 1$
B
$\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i \leq 1 \wedge \alpha_j \leq 1$
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
$0 = 5 \wedge \alpha_j \leq 1 \wedge \alpha_j \leq 1 \wedge \alpha_i = 1 \wedge \boxed{\alpha_j = 0} \Longleftrightarrow false$
OK
$0 = 5 \wedge \alpha_j \leq 1 \wedge \alpha_j \leq 1 \wedge \alpha_i = 0 \wedge \boxed{\alpha_j = 1} \Longleftrightarrow false$
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.

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[1]. 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 `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 $ite(\alpha_j = 0, \sigma(a[0]), \sigma(a[1])) = ite(\alpha_j = 0, ite(\alpha_i = 0, 5, 0), ite(\alpha_i = 1, 5, 0))$.

In general, a symbolic address may reference any cell in memory, making the approaches described above intractable. Fortunately, in many practical cases 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.

To model fully symbolic pointers, 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, which can model operations on arrays as

[1] In propositional logic, the $ite(\mathtt{c}, \mathtt{t}, \mathtt{f})$ expression could be replaced with the formula $(\mathtt{c} \wedge \mathtt{t}) \vee (\neg\mathtt{c} \wedge \mathtt{f})$.

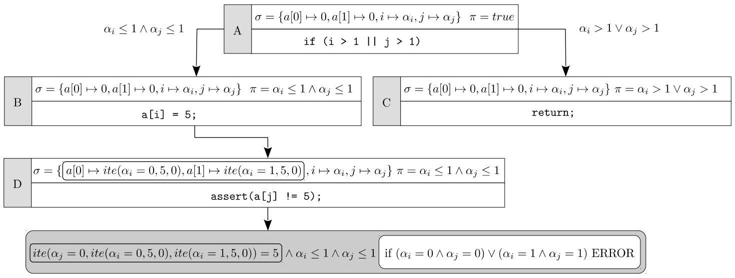

A
$\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = true$
if (i > 1 || j > 1)
$\alpha_i \leq 1 \wedge \alpha_j \leq 1$
$\alpha_i > 1 \vee \alpha_j > 1$
B
$\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i \leq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

first-class entities in constraint formulas using *theories of arrays* in their decision procedures [Ganesh and Dill 2007].

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 is a natural choice for offline executors (Section 2.2) such as DART [Godefroid et al. 2005] and early SAGE releases [Godefroid et al. 2008] that concretely execute one path at a time while collecting path constraints along executed paths. Systems such as CREST [Burnim and Sen 2008] and CUTE [Sen et al. 2005] are capable of reasoning only about equality constraints for pointers, as they can be solved efficiently, and resort to concretization for general symbolic references.

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 region may not be continuous, and the values within the memory range of a symbolic pointer might have structure. MAYHEM [Cha et al. 2012] thus performs a number of optimizations, including Value Set Analysis [Duesterwald 2004] and forms of query caching (Section 7), 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).

3.4. Complex Objects

[Khurshid et al. 2003] introduces a generalization of traditional symbolic execution to advanced constructs of object-oriented languages such as Java and C++. The authors describe a verification framework that combines symbolic execution and model checking to handle dynamically allocated 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 (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. This on-demand concretization enables symbolic execution of methods without requiring an a priori bound on the number of input objects. Also, forking the state as in (2) results into a systematic treatment for aliasing, i.e., when an object can be accessed through multiple references.

[Khurshid et al. 2003; Visser et al. 2004] combine lazy initialization with user-provided *method preconditions*, i.e., conditions which 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.

Further refinements to lazy initialization are described in a number of works, e.g., [Deng et al. 2012; Geldenhuys et al. 2013; Rosner et al. 2015], which all share the goal of reducing the number of heap configurations to generate when forking the state. [Deng et al. 2012] also provides a formal treatment of lazy initialization in Java.

Of a different flavor is the technique presented in [Shannon et al. 2007] for symbolic execution over objects instantiated from commonly used libraries. The authors argue that performing symbolic execution at the representation level might be redundant if the aim is to only check the client code, thus trusting the correctness of the library implementation. They discuss the idea of symbolically executing methods of the Java `String` class using a finite-state automaton that abstracts away the implementation details. They present a case study of an application that dynamically generates SQL queries: symbolic execution is used to check whether the statements conform to the SQL grammar and possibly match injection patterns. The authors mention that their approach might be used to symbolically execute over standard container classes such as trees or maps.

4. INTERACTION WITH THE ENVIRONMENT

When a program interacts with its environment – e.g., file system, environment variables, network - a symbolic executor has to take into account the whole software stack surrounding it, including system libraries, kernel, drivers, etc.

A body of early works (e.g., DART [Godefroid et al. 2005], CUTE [Sen et al. 2005], and EXE [Cadar et al. 2006]) includes the environment in symbolic 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 also results in having calls from distinct execution paths interfere with each other.

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

Fig. 8: Loop example with input read from the environment [Cadar and Sen 2013].

Another way to tackle the problem is to create an abstract model that captures these interactions. For instance, in KLEE [Cadar et al. 2008] symbolic files are supported through a simple *symbolic file system*, which is private for each execution state. In particular, it consists 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 an error-prone and rather expensive process [Ball et al. 2006], models are generally implemented at system call-level rather than library level. An additional benefit of this approach is that it allows for symbolic exploration of the libraries as well.

AEG [Avgerinos et al. 2011] models most of the system environment that an attacker can possibly use as an 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 further extends support to many other POSIX libraries, allowing the user to also control advanced conditions in the testing environment. For instance, CLOUD9 is able to simulate delay, reordering, and dropping of packets 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 also lose it as the modeled system evolves. It would be thus preferable to have analyzed programs to interact with the real environment while exploring multiple paths. In their S^{2}E platform the authors rely on virtualization to perform the desired analysis on the real software stack, preventing side effects from propagating across independent execution paths.

The approach followed in DART [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. Library functions are normally not considered external functions as they are controlled by the program, but in practice the user can adjust the boundary between library and external functions to simulate the desired effects.

5. LOOPS

Loops are one of the main causes of path explosion: 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.

Example. Consider the code fragment of Figure 8 [Cadar and Sen 2013], 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.

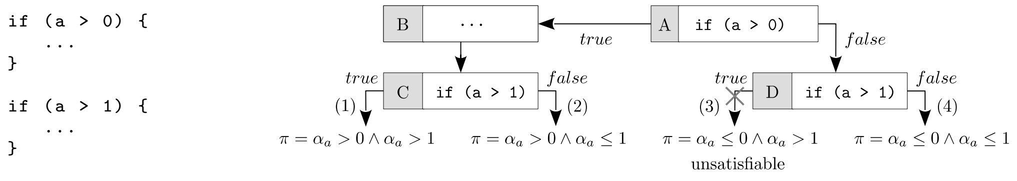
The problem of path explosion due to symbolic execution of loops has been attacked from different sides. A first natural strategy adopted by many symbolic engines is to limit the loop exploration up to a certain number of iterations. Obviously, this may lead to missing interesting paths in the program. For this reason, some works (e.g., AEG [Avgerinos et al. 2011]) have also considered the opposite strategy, allowing the engine to fully explore some loops. To mitigate the path explosion problem, only a single instance of the symbolic executor is allowed to fully unroll a loop, while other instances conservatively explore other paths. This approach has been shown to be effective in some application contexts such as security (e.g., identification of buffer overflows) where interesting behavior may be observed at the loop boundaries.

By using static or dynamic analysis techniques, it may be possible to derive properties over a loop that can be exploited by the symbolic engine to significantly prune branching paths. For instance, knowledge of the exact number of loop iterations - or at least a constant upper bound on it - can significantly help the engine. Section 6.4 provides a more general discussion of how preconditions can help symbolic execution. Nevertheless, even symbolic execution can be used to derive loop invariants. Indeed, if a program contains an assertion after the loop, the approach presented in [Păsăreanu and Visser 2004] works backwards from the property to be checked and it iteratively applies approximation to derive loop invariants. The main idea is to pick the asserted property as the initial invariant candidate and then to exploit symbolic execution to check whether this property is inductive. If the invariant cannot be verified for some loop paths, it is replaced by a different invariant. The next candidate for the invariant is generated by exploiting the path constraints for the paths on which the verification has failed. Additional refinements steps are performed to guarantee termination.

[Godefroid and Luchaup 2011] presents a technique that automatically derives partial summarizations for loops. A loop summarization is similar to a function summary (Section 2.3), using a set of preconditions and a set of postconditions. These are computed dynamically during the symbolic execution by reasoning on the dependencies among loop conditions and symbolic variables. As soon as a loop summary is computed, it is cached for possibly subsequent reuse. This not only allows the symbolic engine to avoid redundant executions of the same loop under the same program state, but also makes it possible to generalize the loop summary to cover even different executions of the same loop that run under different conditions. A main limitation of this approach is that it can generate summaries only for loops that iteratively update symbolic variables across loop iterations by adding a constant, non-zero amount.

[Slaby et al. 2013] introduces a technique of a different flavor that analyzes cyclic paths in the control flow graph of a given program and produces *templates* that declaratively describe the program states generated by these portions of code into a symbolic execution tree. By exploiting templates, the symbolic execution engine needs to 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.

It has also been observed that loop executions may strictly depend on input features. *Loop-extended symbolic execution* [Saxena et al. 2009] is able to effectively explore



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

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

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

(a) (b)

Fig. 9: Pruning unrealizable paths example: (a) Fragment of code; (b) Symbolic execution of the code: the *true* branch in node D is not explored since its path constraints $(\alpha_a \leq 0 \wedge \alpha_a > 1)$ are not satisfiable, i.e., there exists no assignment of variable a that can drive a concrete execution first toward node D and then through its *true* branch.

a loop whenever a grammar describing the input program is available. Relating the number of iterations with features of the program input can guide the exploration of the program states generated by a loop.

6. PATH EXPLOSION

One of the main challenges of symbolic execution is the path explosion problem. Since symbolic execution may fork off a new execution engine instance at every branch, the total number of executors may be exponential in the number of branches in the program. This impacts both time and space, as a symbolic executor may need to keep track of an exponential number of pending branches to be explored. A common approach is to compute an under-approximation of the analysis that only explores a relevant subset of the state space.

6.1. Pruning Unrealizable Paths

A first natural technique for reducing the path space is invoking the constraint solver at each branch, pruning branches that are not realizable. Indeed, if the constraint solver is able to prove that the logical formula given by the path constraints of a branch is not satisfiable, then there exists no assignment of the program input values that would drive a real execution toward that path. For this reason, the symbolic engine can safely discard the path involving that branch without affecting soundness of the approach.

Example. Consider the example shown in Figure 9. Assuming that a is a local variable bound to an unconstrained symbol α_a, a symbolic engine would start the execution of the code fragment in Figure 9a by evaluating the branch condition $a > 0$. Before expanding both branches, the symbolic engine queries a constraint solver to verify that no contradiction arises when adding to the path constraints π the *true* branch condition $(\alpha_a > 0)$ or the *false* branch condition $(\alpha_a \leq 0)$. Since both paths are feasible, the engine forks the execution states B and D (Figure 9b). A similar scenario happens when the engine evaluates the branch condition $a > 1$. However, since α_a is not unconstrained anymore, some contradictions may be actually possible. The engine queries the solver to check the following path constraints: (1) $\alpha_a > 0 \wedge \alpha_a > 1$, (2) $\alpha_a > 0 \wedge \alpha_a \leq 1$, (3) $\alpha_a \leq 0 \wedge \alpha_a > 1$, and (4) $\alpha_a \leq 0 \wedge \alpha_a \leq 1$. Notice that formula $\alpha_a \leq 0 \wedge \alpha_a > 1$ does not admit a valid solution and thus the related path can be safely dropped by the engine since it is unrealizable. On the other hand, other paths admit a valid solution and can be further explored by the engine.

This approach is commonly referred as *eager evaluation* of path constraints since path constraints are eagerly checked at each branch and is typically the default approach

Heuristic	**Goal**
BFS	*Maximize coverage* [Chipounov et al. 2012; Tillmann and De Halleux 2008]
DFS	*Exhaust paths, minimize memory usage* [Cadar et al. 2006; Chipounov et al. 2012] [Tillmann and De Halleux 2008; Godefroid et al. 2005]
Random path selection	*Randomly pick a path with probability based on its length* [Cadar et al. 2008]
Code coverage search	*Prioritize paths that may explore unexplored code* [Cadar et al. 2006; Cadar et al. 2008; Cha et al. 2012] [Chipounov et al. 2012; Groce and Visser 2002]
Buggy-path-first	*Prioritize bug-friendly path* [Avgerinos et al. 2011]
Loop exhaustion	*Fully explore specific loops* [Avgerinos et al. 2011]
Symbolic instruction pointers	*Prioritize paths with symbolic instruction pointers* [Cha et al. 2012]
Symbolic memory accesses	*Prioritize paths with symbolic memory accesses* [Cha et al. 2012]
Fitness function	*Prioritize paths based on a fitness function* [Xie et al. 2009; Cadar and Sen 2013; Xie et al. 2009]
Subpath-guided search	*Use frequency distributions of explored subpaths to prioritize less covered parts of a program* [Li et al. 2013]

Fig. 10: Common path selection heuristics discussed in literature.

adopted by most symbolic engines. We refer to Section 7 for a discussion of the possible benefits given by the opposite strategy, i.e., *lazy evaluation*, where path constraints are lazily checked in order to possibly reduce the burden on the solver.

6.2. Bounding Computational Resources

Another common approach is to limit the amount of resources symbolic execution is allowed to use. For instance, the computation may time out after a certain amount of time. Since only a fraction of paths may be explored, the search should be prioritized by looking at the most promising paths first. There are several strategies for selecting or generating the next path to be explored. We now briefly overview some of the most interesting techniques that have been shown to be effective in the literature.

Search Heuristics. Given a set of unexplored paths, a search heuristic should select the most promising path to explore. Many works have introduced novel search strategies, showing their effectiveness in specific application contexts. These heuristics have often been tailored to help the symbolic engine achieve a specific goal (e.g., overflow detection). Finding a universally optimal strategy for prioritizing path exploration remains an open problem. Table 10 provides a sample of prominent search heuristics discussed in prior works.

The most common strategies are *depth-first search* (DFS) and *breadth-first search* (BFS). DFS continuously expands a path as much as possible, before backtracking to the deepest unexplored branch. BFS explores all unexplored paths in parallel, repeatedly expanding each of them by a fixed slice. DFS is often adopted for minimizing the memory usage of the symbolic engine: since the chosen path will be sooner or later fully explored, the memory needed for keeping its state will be released as well. Unfortunately, paths containing loops and recursive calls can easily stall the symbolic engine. For this reason, some tools prefer prioritizing paths using BFS. Although memory pressure can be higher, this strategy may allow an engine to quickly explore diverse paths and possibly detecting interesting behaviors. However, if the ultimate goal requires to fully terminate the exploration of one or more paths, BFS may take a very long time.

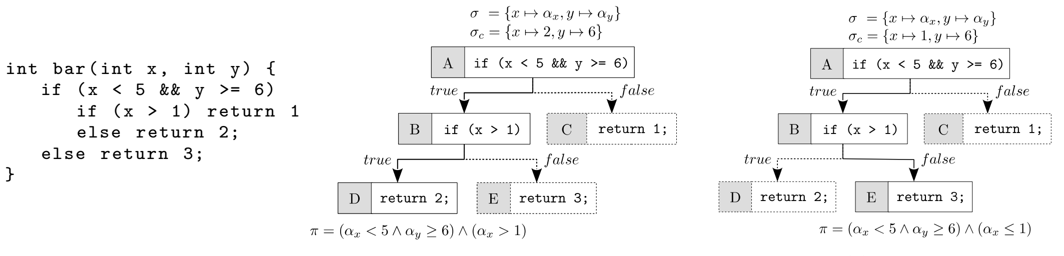
Another popular strategy is *random path selection* that, as its name would suggest, randomly picks a path for exploration. This heuristic has been refined in several variants. For instance, KLEE [Cadar et al. 2008] assigns probabilities to paths based on their length and on the branch arity. Namely, 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^2E [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 a weight for each state and then randomly selects a state according to these weights. The weight is obtained by taking into account the minimum distance to an uncovered instruction, the call stack of the state, and whether the state has recently covered new code. 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.

Other search heuristics try to prioritize paths likely leading to states that are *interesting* according to some goal. For instance, the *buggy-path first* strategy in AEG [Avgerinos et al. 2011] 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 discussed in AEG [Avgerinos et al. 2011] 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 symbolic memory accesses are identified or symbolic instruction pointers are detected.

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 papers, e.g., [Xie et al. 2009; Cadar and Sen 2013; Xie et al. 2009], have applied this idea in the context of symbolic execution. As an example, [Xie et al. 2009] introduces *fitnex*, a strategy for concolic execution that prioritizes paths that are *closer* to take a specific branch. In more detail, given a branch condition of the form $|a - c| == 0$ and a path that has reached the branch, *fitnex* computes a closeness equal to $|a - c|$ 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.

Dynamic Test Generation. Traditional symbolic execution does not scale over large programs. Although search heuristics may help prioritize some interesting paths, symbolic execution may still proceed extremely slow. Indeed, the engine must simulate any instruction in the program and heavily relies on the constraint solver in order to make any progress in the execution. *Dynamic test generation*, initially introduced in DART [Godefroid et al. 2005], is a technique that can help symbolic execution scale to large programs. The main idea is to execute a program both concretely and symbolically. This kind of execution is often referred to as concolic execution (Section 2.1). Initially, a random input is generated and a concrete execution is started. In parallel, a symbolic execution is also started. 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.



```
int bar(int x, int y) {
   if (x < 5 && y >= 6)
      if (x > 1) return 1
      else return 2;
   else return 3;
}
```

$\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$
$\sigma_c = \{x \mapsto 2, y \mapsto 6\}$
A if (x < 5 && y >= 6)
true
false
B if (x > 1)
C return 1;
true
false
D return 2;
E return 3;
$\pi = (\alpha_x < 5 \wedge \alpha_y \geq 6) \wedge (\alpha_x > 1)$
$\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$
$\sigma_c = \{x \mapsto 1, y \mapsto 6\}$
A if (x < 5 && y >= 6)
true
false
B if (x > 1)
C return 1;
true
false
D return 2;
E return 3;
$\pi = (\alpha_x < 5 \wedge \alpha_y \geq 6) \wedge (\alpha_x \leq 1)$

(a) (b) (c)

Fig. 11: Dynamic test generation example. (a) Source code of function `bar`. (b) Symbolic execution tree for the function `bar`. Solid edges show the path taken by the concolic execution when `x = 2` and `y = 6`. These input values have been randomly chosen. (c) Concolic execution when `x = 1` and `y = 6`. These input values have been obtained using a constraint solver, after negating the path constraints of node B.

In other words, the symbolic execution is driven by a specific concrete execution. A consequence of this approach is that the symbolic engine does not need to invoke the constraint solver to decide whether a branch condition is (un)satisfiable, since this is directly tested by the concrete execution. Other paths of the program execution can be then explored by selecting a previously taken branch and negating its constraints. Using a constraint solver, the symbolic engine can generate a new (random) input that drives the concrete execution toward a new path. This strategy can be repeated as much as needed to achieve the desired coverage.

Example. An example of dynamic test generation is shown in Figure 11. Consider the function `bar` (Figure 11a) that takes two integer inputs `x` and `y`. To start a first concrete execution, a symbolic engine may initially randomly pick `x = 2` and `y = 6` as input values. The concrete execution induced by these inputs is presented in Figure 11b: both the first and second branch condition (nodes A and B) are satisfied, directing the execution toward the first `return` statement (node D). Nodes C and E are skipped since their associated branch conditions are not met by the current input values. For instance, node E is not executed since the condition $x > 1$ (node B) directs the path toward the node D. An engine that desires to symbolically execute a path containing the node E must track the constraints during the concrete execution over `x = 2` and `y = 6` and then negate the branch condition $x > 1$. To generate a new input, the engine then invokes a solver over the constraints $\neg(\alpha_x > 1) \wedge (\alpha_x < 5 \wedge \alpha_y > 5)$, getting, e.g., `x = 1` and `y = 6` (Figure 11c). Notice that since `y` is not involved in the branch condition that is currently negated, the engine may reuse its value and include an additional constraint $\alpha_y = 6$. This optimization may drastically reduce the solving time required to obtain a solution from the constraint solver.

Although dynamic test generation 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 the performed concrete executions can be very large, the search heuristics discussed in Section 6.2 still play a crucial role. For instance, DART [Godefroid et al. 2005] chooses the next branch to negate using a DFS strategy. Additional strategies for picking the next branch to negate have been presented in literature. For instance, the *generational search* algorithm discussed in 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* and, for this reason, symbolic engines such as SAGE are often referred as *white-box fuzzers*.

6.3. Under-Constrained Symbolic Execution

A possible approach to avoid path explosion is to cut the code to check, 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 has proven to be quite difficult due to the entanglements with the surrounding environment [Engler and Dunbar 2007].

The main issue is that 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., Check ’n’ Crash [Csallner and Smaragdakis 2005], first analyze the code in isolation and then test the generated crashing inputs using concrete executions.

Under-constrained symbolic execution [Engler and Dunbar 2007] is a twist on symbolic execution that allows for the analysis of a function in isolation by marking some symbolic inputs 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 to analyze. Under-constrained variables have the same semantics as classic symbolic variables except when used in an expression that can cause an error to occur. 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 choice 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.

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 affected by it as under-constrained.

6.4. Preconditioned Symbolic Execution

AEG [Avgerinos et al. 2011] proposes *preconditioned symbolic execution* as a novel method to drive symbolic execution toward certain subsets of the input space. A subset is determined by a precondition predicate: inputs that do not satisfy it will not be explored. The intuition behind preconditioned symbolic execution is that we can narrow down the state space we are exploring by specifying goal-oriented (e.g., exploitability) conditions as a precondition, e.g., all symbolic inputs should have the maximum size to expose buffer overflow bugs. The main benefit of preconditioned symbolic execution is simple: by limiting the size of the input state space before execution begins, we can prune program paths and therefore explore the target program more efficiently. Indeed, preconditions need to be selected carefully. If a precondition is too specific, no bug may be detected; if it is too general, almost the entire state space will be explored.

This technique enforces a precondition by adding the precondition constraints to the path predicate during initialization. Adding constraints may seem counterintuitive since there are more checks to perform at branch points during symbolic execution. However, the state space shrink caused by the precondition constraints outweighs the

```
// N symbolic branches
if (input[0] < 42) [...]
[...]
if (input[N-1] < 42) [...]

// symbolic loop
strcpy(dest, input);

// M symbolic branches
if (input[N] < 42) [...]
[...]
if (input[N+M-1] < 42) [...]
```

(a)

```
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.  }
```

(b)

Fig. 12: (a) Preconditioned symbolic execution example [Avgerinos et al. 2011]; (b) State merging example

decision procedure overhead at branching points. When the precondition for a branch is unsatisfiable, neither further checks are made, nor a new execution engine instance is forked for the branch.

Four prominent categories of preconditions are:

— *None*: there is no precondition, thus space exploration proceeds as normal;
— *Known length*: symbolic inputs are of known maximum length, e.g., a network packet has a fixed size, or static analysis can determine the input length;
— *Known prefix*: symbolic inputs have a known prefix, e.g., a fixed header string such as the initial *magic code* in a binary, or a network packet header.
— *Fully known*: all input bytes are concrete, as in a concolic execution; it can be used, for instance, to generate a working exploit from a known crashing input.

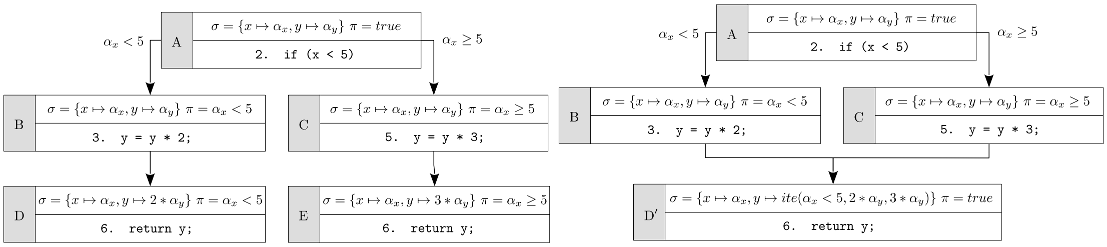
Example. Consider the code fragment in Figure 12a where `input` is an array of $s \geq n + m$ bytes. The impact of preconditions on the state space size is as follows:

— *None*: the input space size is 256^s, and up to $2^n \cdot s \cdot 2^m$ execution engine instances are needed, due to $n + m$ symbolic branches and up to S loop iterations;
— *Known length*: if we assume a string length s, i.e., the first $(s - 1)$ bytes of `input` are not $\backslash 0$, the loop is concretized, and the state space size is reduced to 2^{n+m};
— *Known prefix*: if a prefix of $p < n$ bytes is known for `input`, the first p branches and p loop iterations are concrete, and the state space size becomes $2^{n-p} \cdot s \cdot 2^m$;
— *Fully known*: as all input bytes are concrete, the state space size is trivially 1.

6.5. State Merging

Several static program analysis techniques such as abstract interpretation merge states corresponding to different paths into a state that over-approximates them. In a precise symbolic execution, however, merging is not allowed to introduce any approximation or abstraction, and therefore can only change formulas to have them characterize sets of execution paths. In other words, a merged state will be described by a formula that represents the disjunction of the formulas that would have described the individual states if they were kept separate.

Example. Consider the function of Figure 12b and its symbolic execution tree shown in Figure 13a. 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. Line 2 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


$\alpha_x < 5$
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;
D′
$\sigma = \{x \mapsto \alpha_x, y \mapsto ite(\alpha_x < 5, 2 * \alpha_y, 3 * \alpha_y)\}$ $\pi = true$
6. return y;

(a) (b)

Fig. 13: Symbolic execution of function foo of Figure 12b: (a) without state merging; (b) with state merging.

symbol α_x (execution states C and D, respectively). After expanding every execution state until the return at line 6 is reached on all branches, the symbolic execution tree gets populated with two additional states D and E. If a symbolic execution engine desires to reduce the number of active states, then state merging can be performed. For instance, Figure 13b shows the symbolic execution DAG for the same piece of code when a state merging operation is performed before evaluating the return statement at line 6: D' is now 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'.

Trade-Offs: to Merge or Not to Merge? 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 typically encounter difficulties when dealing with disjunction. Merging can also introduce new symbolic expressions in the code, for instance 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 the paths does not perform any merge and is used in search-based symbolic execution (Section 6.2). At the other end, static state merging combines states at control-flow join points, thus 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 by, e.g., 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. Hard statements are those that involve system calls, indirect jumps, and other operations that are difficult to reason about statically. 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. Ideally, in order to maximize the opportunities for merging, a symbolic execution 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 aiming at prioritizing more "interesting" states over others. [Kuznetsov et al. 2012] introduces *dynamic state merging* to identify opportunities for merging 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.

6.6. Leveraging Program Analysis and Optimization Techniques

A deeper understanding of a program's behavior can help a symbolic engine to 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. Some prominent examples are listed below:

— *Program slicing* is a method that, 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, given a program slice related to a target program point, symbolic exploration can be restricted to paths contained in the program slice. We discuss an example of use in Section 9.3.

— *Taint analysis* attempts to identify variables that can be modified by predefined taint sources such as user input and can be performed both statically and dynamically. Dynamic taint analysis typically yields more accurate results, and is often employed to explore symbolically only the parts of an execution that depend upon tainted values [Schwartz et al. 2010].

— *Fuzzing* is a software testing technique that randomly mutates user-provided test inputs to cause crashes or assertion failures and find potential memory leaks. Fuzzing, as discussed in Section 6.2, 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: we present two embodiments of this approach in Section 9.1.

— *Branch predication* 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.

— *State matching* determines whether an abstract state is subsumed by another, and can be used to analyze an under-approximation of a program's behavior. For instance, [Anand et al. 2006; Visser et al. 2006] explore different heap shapes in the context of test generation for data structures, using subsumption checking to determine whether a symbolic state is being revisited.

— *Type checking* can be effectively mixed with symbolic analysis [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[2].

7. 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].

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 inequalities 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.

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. The two most popular solvers used in symbolic executors are STP and Z3. STP [Ganesh and Dill 2007; Ganesh 2007] is an SMT solver with bitvector and array theories initially developed at Stanford and employed in, e.g., EXE [Cadar et al. 2006], KLEE [Cadar et al. 2008], MINESWEEPER [Brumley et al. 2008], and AEG [Avgerinos et al. 2011]. Z3 [De Moura and Bjørner 2008] is an SMT solver developed at Microsoft with support for nonlinear arithmetic, bitvector, and array theories, and is used in, e.g., MAYHEM [Cha et al. 2012], SAGE [Godefroid et al. 2012], and ANGR [Shoshitaishvili et al. 2016]. CVC3 [Barrett and Tinelli 2007] is another SMT solver that supports theories for linear arithmetic, bitvectors, arrays, and quantifiers, and is employed in JAVA PATHFINDER [Păsăreanu and Rungta 2010] along with CHOCO [Prud'homme et al. 2015] for integer/real constraints and CORAL [Souza et al. 2011] for complex mathematical constraints. Modern symbolic executors can typically choose between different underlying solvers through a common API, and also resort to a native interface to a specific solver for better performance.

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.

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. Two prominent approaches consist in: (i) reducing the size and complexity of the constraints to check, and (ii) unburdening the solver by, e.g., resorting to constraint solution caching, deferring of solver queries, or concretization.

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]).

[2] Interestingly, [Khoo et al. 2010] discusses also how symbolic analysis can help a type checker. For instance, a symbolic engine can provide context-sensitive properties over a variable that would rule out certain type errors, improving the precision of the type checker.

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, real programs typically have many independent branches, which can often introduce irrelevant constraints that add up quickly.

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. As 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 are masked away by bit operations. For instance, if x is a 4-bit symbolic value, the most significant bit in the expression $x \,|\, 1000$ is known to be one. The simplifier can propagate information across the tree representation of an expression, and if all bits in an expression are known, it replaces the expression 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 syntactic or semantic equivalence of the constraints.

EXE [Cadar et al. 2006] caches the results of satisfiability queries and constraint solutions in order to avoid calling the solver when possible. A cache is managed 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*. A tree representation-based cache maps constraint sets 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] introduces a new approach for a more efficient application of symbolic execution. The work is motivated by the observation that applying symbolic execution often requires several successive runs of the technique on largely similar underlying problems, e.g., finding a bug and then examining the program again to check the validity of the fix. A trie-based data structure compactly encodes the choices taken when exploring different paths, allowing successive runs to reuse previously computed results where possible.

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 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

```
1. int non_linear(int v) {
2.     return (v*v) % 50;
3. }

4. void test(int x, int y) {
5.     z = non_linear(y);
6.     if (z == x) {
7.        if (x > y + 10) ERROR;
8.     }
9. }
```

Fig. 14: Example with non-linear constraints.

supports constraint reuse based on the logical implication relations among constraints, leading to better reuse and faster execution time for the symbolic analysis.

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 as unreachable in a real execution.

Compared to the *eager* approach of checking the feasibility of a branch as encountered (Section 6.1), 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 cannot be (efficiently) solved by constraint solvers.

Example. In the code fragment of Figure 14, the engine stores a non-linear constraint of the form $\alpha_x = (\alpha_y * \alpha_y)\,\%\,50$ for the $true$ branch at line 6. A solver that does not support non-linear arithmetic fails to generate any input for the program.

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 sacrificing soundness in the exploration. For instance, in the presence of three nested branches with only one being nonlinear, DART [Godefroid et al. 2005] starts from a random valid input for the function, and then alters it when symbolically exploring the two linear branches. The work resorts to concretization also to avoid performing expensive or imprecise alias analysis on pointers.

To partially overcome the incompleteness due to concretization, [Păsăreanu et al. 2011] suggests to consider *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 [Păsăreanu et al. 2011] requires instead to detect *solvable* constraints along a full path and to delay concretization as much as possible.

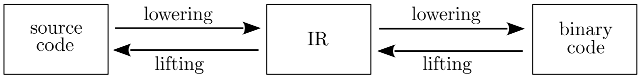


Fig. 15: Lowering and lifting processes in native vs. source code processing.

8. SYMBOLIC EXECUTION OF BINARY CODE

The importance of performing symbolic analysis of program properties on binary code is on the rise for a number of reasons. Binary code analysis is attractive as it reasons on code that will actually execute: not requiring the source code significantly extends the applicability of such techniques (to, e.g., common off-the-shelf proprietary programs, firmwares for embedded systems, and malicious software), and it gives the ground truth important for security applications whereas source code analysis may yield misleading results due to compiler optimizations [Song et al. 2008]. Binary analysis is relevant also for programs written in dynamic languages, as nowadays they are executed in modern runtimes that deeply transform and optimize their code before just-in-time compiling it.

Analyzing binary code is commonly seen as a challenging task due to its complexity and lack of a high-level semantics. Modern architectures offer complex instruction sets: modeling each instruction can be difficult, especially in the presence of multiple side effects on processor flags to determine branch conditions. The second major challenge comes from the lack of the higher-level semantics present in source code, especially when no debugging information is available. Types are not explicitly encoded in binary code: even with register types, it is common to store values as one type and read them as another. Similar considerations can be made for array bounds as well. Also, control-flow graph information is not explicitly available, as control flow is performed through jump instructions at both inter- and intra-procedural level. The function abstraction at the binary level does not exist as we intend it at source-code level: functions can be separated in non-contiguous pieces, and code may also call in the middle of a code block generated for a source-level function.

In the remainder of this section we provide an overview of how symbolic executors can address some of the most significant challenges in the analysis of binary code.

8.1. Lifting to an Intermediate Representation

Motivated by the complexity in modeling native instructions and by the variety of architectures on which applications can be deployed (e.g., x86, x86-64, ARM, MIPS), symbolic executors for binary code typically rely on a *lifter* that transforms native instructions into an *intermediate representation* (IR), also known as *bytecode*. Modern compilers such as LLVM typically generate IR by *lowering* the user-provided source code during the first step of compilation, optimize it, and eventually lower it to native code for a specific platform. Source-code symbolic executors can resort to compiler-assisted lowering to reason on bytecode rather than source-language statements: for instance, KLEE [Cadar et al. 2008] reasons on the IR generated by the LLVM compiler for static languages such as C and C++. Figure 15 summarizes the relationships between source code, IR, and binary code.

Reasoning at the intermediate representation level allows program analyses to be written in an architecture-independent fashion. Translated instructions will always expose all the side-effects of a native instruction, and support for additional platforms can be added over time. A number of symbolic executors use VEX, the intermediate representation format from the Valgrind dynamic instrumentation framework. VEX is a RISC-like language designed for program analysis that offers a compact set of instructions for expressing programs in static single assignment (SSA) form. Lifters are available for both 32-bit and 64-bit ARM, MIPS, PPC, and x86 binaries.

ANGR [Shoshitaishvili et al. 2016] performs analysis directly on VEX IR. Authors chose VEX over other IR formats as at that time it was the only choice that offered a publicly available implementation with support for many architectures. Also, they mention that writing a binary lifter can be a daunting task, and a well-documented and program analysis-oriented solution can be a bonus. BITBLAZE [Song et al. 2008] uses VEX too, although it translates it to a custom intermediate language. The reason for this is that VEX captures the side effects of some instructions only implicitly, such as what EFLAGS are set by x86 instructions: translating it to a custom language simplified the development of BITBLAZE's analysis framework.

The authors of S^2E [Chipounov et al. 2012] have implemented an x86-to-LLVM-IR lifter in order to use the KLEE [Cadar et al. 2008] symbolic execution engine for whole-system symbolic analysis of binary code in a virtualized environment. The translation is transparent to both the guest operating system and KLEE, thus enabling the analysis of binaries using the full power of KLEE. Another x86-to-LLVM-IR lifter that can be used to run KLEE on binary code is mcsema[3].

8.2. Reconstructing the Control Flow Graph

A control flow graph (CFG) can provide valuable information for a symbolic executor as it captures the set of potential control flow transfers for all feasible execution paths. A fundamental issue that arises when reconstructing CFGs for binaries is that the possible targets of an indirect jump may not be identified correctly. Direct jumps are straightforward to process: as they encode their targets explicitly in the code, successor basic blocks can be identified and visited until no new edge is found. The target of an indirect jump is determined instead at run time: it might be computed by carrying out a calculation (e.g., a jump table) or depend on the current calling context (e.g., a function pointer is passed as argument, or a virtual C++ method is invoked).

CFG recovery is typically an iterative refinement process based on a number of program analysis techniques. For instance, Value Set Analysis (VSA) [Duesterwald 2004] is a technique that can be used to identify a tight over-approximation of certain program state properties (e.g., the set of possible targets of an indirect jump or a memory write). In BITBLAZE [Song et al. 2008] an initial CFG is generated by inserting special successor nodes for unresolved indirect jump targets. This choice is conceptually similar to widening a fact to the bottom of a lattice in a data-flow analysis. When an analysis requires more precise information, VSA is then applied on demand.

ANGR [Shoshitaishvili et al. 2016] implements two algorithms for CFG recovery. An iterative algorithm starts from the entry point of the program and interleaves a number of techniques to achieve speed and completeness, including VSA, inter-procedural backward program slicing, and symbolic execution of blocks. This algorithm is however rather slow and may miss code portions reachable only through unresolved jump targets. The authors thus devise a fast secondary algorithm that uses a number of heuristics to identify functions based on prologue signatures, and performs simple analyses (e.g., a lightweight alias analysis) to solve a number of indirect jumps. The algorithm is context-insensitive, so it can be used to quickly recover a CFG without a concern for understanding the reachability of functions from one another.

8.3. Code Obfuscation

In recent years, code obfuscation has received considerable attention as a cheap way to hinder the understanding of the inner workings of a proprietary program. Obfuscation is employed not only to thwart software piracy and improve software security, but

[3] https://github.com/trailofbits/mcsema.

also to avoid detection and resist analysis for malicious software [Udupa et al. 2005; Yadegari et al. 2015].

A significant motivation behind using symbolic/concolic execution in malware analysis is to get around code obfuscations. However, current analysis techniques have trouble dealing with some of those obfuscations, leading to imprecision and/or excessive resource usage [Yadegari and Debray 2015]. For instance, obfuscation tools can transform conditional branches into indirect jumps that symbolic analysis find difficult to analyze, while run-time code self-modification might conceal conditional jumps on symbolic values so that they are missed by the analysis.

A few works have described obfuscation techniques aiming at thwarting symbolic execution. [Sharif et al. 2008] introduces a *conditional code obfuscation* scheme based on one-way hash functions that makes it hard to identify the values of symbolic variables for which branch conditions are satisfied. They also present an encryption scheme for the code to execute based on a key derived from the value that satisfies a branch condition. [Wang et al. 2011] takes a step forward by proposing an obfuscation technique that works despite it uses linear operations only, for which symbolic execution usually works well. The obfuscation tool inserts a simple loop incorporating an unsolved mathematical conjecture that converges to a known value after a number of iterations, and the produced result is then combined with the original branch condition.

[Hai et al. 2016] presents BE-PUM, a tool to generate a precise CFG in the presence of obfuscation techniques that are common in the malware domain, including indirect jumps, structured exception handlers (SEHs), overlapping instructions, and self-modifying code. While engines such as BITBLAZE [Song et al. 2008] typically rely on disassemblers like IDA Pro[4], BE-PUM relies on concolic execution to deobfuscate code, using a binary emulator for the user process and stubs for API calls.

[Yadegari and Debray 2015] discusses the limitations of symbolic execution in the presence of three generic obfuscation techniques: (1) conditional-to-indirect jump transformation, also known as *symbolic jump problem* [Schwartz et al. 2010]; (2) conditional-to-conditional jump transformation, where the predicate is deeply changed; and (3) symbolic code, when code modification is carried out using an input-derived value. The authors show how resorting to bit-level taint analysis and architecture-aware constraint generation can allow symbolic execution to circumvent such obfuscations.

9. SAMPLE APPLICATIONS

The last decade has witnessed an increasing adoption of symbolic execution techniques not only in the software testing domain, but also to address other compelling engineering problems such as automatic generation of exploits or authentication bypass. We now discuss prominent applications of symbolic execution techniques to these domains. Examples of extensions to other areas can be found, e.g., in [Cadar et al. 2011].

9.1. Bug Detection

Software testing strategies typically attempt to execute a program with the intent of finding bugs. As manual test input generation is an error-prone and usually non-exhaustive process, automated testing technique have drawn a lot of attention over the years. Random testing techniques such as fuzzing are cheap in terms of run-time overhead, but fail to obtain a wide exploration of a program state space. Symbolic and concolic execution techniques on the other hand achieve a more exhaustive exploration, but they become expensive as the length of the execution grows: for this reason, they usually reveal shallow bugs only.

[4] https://www.hex-rays.com/products/ida/.

[Majumdar and Sen 2007] proposes *hybrid concolic testing* for test input generation, which combines the ability of random search to reach deep program states with the ability of concolic execution to achieve a wide exploration. The two techniques are interleaved: in particular, when random testing saturates (i.e., it is unable to hit new code coverage points after a number of steps), concolic execution is used to mutate the current program state by performing a bounded depth-first search for an uncovered coverage point. For a fixed time budget, the technique outperforms both random and concolic testing in terms of branch coverage. The intuition behind this approach is that many programs show behaviors where a state can be easily reached through random testing, but then a precise sequence of events - identifiable by a symbolic engine - is required to hit a specific coverage point.

[Stephens et al. 2016] refines this idea and devises a vulnerability excavation tool based on Angr [Shoshitaishvili et al. 2016], called Driller, that interleaves fuzzing and concolic execution to discover memory corruption vulnerabilities. The authors remark that user inputs can be categorized as *general* input, which has a wide range of valid values, and *specific* input: a check for particular values of a specific input then splits an application into *compartments*. Driller offloads the majority of unique path discovery to a fuzzy engine, and relies on concolic execution to move across compartments. During the fuzzy phase, Driller marks a number of inputs as interesting (for instance, when an input was the first to trigger some state transition) and once it gets stuck in the exploration, it passes the set of such paths to a concolic engine, which preconstraints the program states to ensure consistency with the results of the native execution. On the dataset used for the DARPA Cyber Grand Challenge qualifying event, Driller could identify crashing inputs in 77 applications, including both the 68 and 16 applications for which fuzzing and symbolic execution alone succeeded, respectively. For 6 applications, Driller was the only one to detect a vulnerability.

9.2. Bug Exploitation

Bugs are a consequence of the nature of human factors in software development and are everywhere. Those that can be exploited by an attacker should normally be fixed first: systems for automatically and effectively identifying them are thus very valuable.

AEG [Avgerinos et al. 2011] employs preconditioned symbolic execution to analyze a potentially buggy program in source form and look for bugs amenable to stack smashing or return-into-libc exploits [Pincus and Baker 2004], which are popular control hijack attack techniques. The tool augments path constraints with exploitability constraints and queries a constraint solver, generating a concrete exploit when the constraints are satisfiable. The authors devise the *buggy-path-first* and *loop-exhaustion* strategies discussed in Section 6.2 to prioritize paths in the search. On a set of 14 Linux applications, AEG could construct 16 control hijack exploits, 2 of which against previously unknown vulnerabilities.

Mayhem [Cha et al. 2012] takes another step forward by presenting the first end-to-end exploitable bug finding system working on programs in binary form. It adopts a hybrid execution model based on checkpoints and two components: a concrete executor that injects taint-analysis instrumentation in the code and a symbolic executor that takes over when a tainted branch or jump instruction is met. Exploitability constraints for symbolic instruction pointers and format strings are generated, targeting a wide range of exploits, e.g., SEH-based and jump-to-register ones. Three path selection heuristics help prioritizing paths that are most likely to contain a bug (e.g., those containing symbolic memory accesses or instruction pointers). A virtualization layer intercepts and emulates all the system calls to the host OS, while preconditioned symbolic execution can be used to reduce the size of the search space. Also, restricting symbolic execution to tainted basic blocks only gives very good speedups in this setting, as in the reported experiments more than 95% of the processed instructions were not tainted. MAYHEM was able to find exploitable vulnerabilities in the 29 Linux and Windows applications considered in the evaluation, 2 of which were previously undocumented. Although the goal in MAYHEM is informing the user that an exploitable bug exists, the generated simple exploits can be likely transformed in an automated fashion to work in the presence of OS defenses such as address space layout randomization and data execution prevention [Schwartz et al. 2011].

9.3. Authentication Bypass

Software backdoors are a method of bypassing authentication in an algorithm, a software product, or even in a full computer system. Although sometimes these software flaws are injected by external attackers using subtle tricks such as compiler tampering [Karger and Schell 1974], there are reported cases of backdoors that have been surreptitiously installed by the hardware and/or software manufacturers [Costin et al. 2014], or even by governments [Zitter 2013].

Different works [Davidson et al. 2013; Zaddach et al. 2014; Shoshitaishvili et al. 2015] have exploited symbolic execution for analyzing the behavior of binary firmwares. Indeed, an advantage of this technique is that it can be used even in environments, such as embedded systems, where the documentation and the source code that are publicly released by the manufacturer are typically very limited or none at all. For instance, [Shoshitaishvili et al. 2015] proposes Firmalice, a binary analysis framework based on ANGR [Shoshitaishvili et al. 2016] that can be effectively used for identifying authentication bypass flaws inside firmwares running on devices such as routers and printers. Given a user-provided description of a privileged operation in the device, Firmalice identifies a set of program points that, if executed, forces the privileged operation to be performed. The program slice that involves the privileged program points is then symbolically analyzed using ANGR. If any such point can be reached by the engine, a set of concrete inputs is generated using an SMT solver. These values can be then used to effectively bypass authentication inside the device. On three commercially available devices, Firmalice could detect vulnerabilities in two of them, and determine that a backdoor in the third firmware is not remotely exploitable.

10. CONCLUSIONS

Techniques for symbolic execution have evolved significantly in the last decade, leading to major practical breakthroughs. In 2016, DARPA has challenged the global innovation community with a $2M prize to build a computer that can hack and patch unknown software with no one at the keyboard. The winner, MAYHEM [Cha et al. 2012], was also the first autonomous computer system to play the Capture-The-Flag contest at the DEF CON 24 hacker convention[5]. The event demonstrated that tools for automatic exploit detection based on symbolic execution can be competitive with human experts, paving the road to unprecedented applications and the rise of start-ups that have the potential to shape software security and reliability in the next decades.

This survey has discussed some of the key aspects and challenges of symbolic execution, presenting them for a broad audience. To explain the basic design principles of symbolic executors and the main optimization techniques, we have focused on single-threaded applications with integer arithmetic. Symbolic execution of multi-threaded programs is treated, e.g., in [Farzan et al. 2013; Bergan et al. 2014; Guo et al. 2015], while techniques for programs that manipulate floating point data are addressed in, e.g., [Barr et al. 2013; Collingbourne et al. 2014; Ramachandran et al. 2015].

We hope that this survey will help non-experts grasp the key inventions in the exciting line of research of symbolic execution, inspiring further work and new ideas.

[5] https://www.defcon.org/html/defcon-24/dc-24-ctf.html.

REFERENCES

Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2006. Symbolic Execution with Abstract Subsumption Checking. In *Proceedings of the 13th International Conference on Model Checking Software (SPIN 2006)*. Springer-Verlag, Berlin, Heidelberg, 163–181.

Athanasios 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS 2011)*.

Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1083–1094.

Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and Precise Extended Static Checking. In *Proceedings of the 30th Intern. Conf. on Software Engineering (ICSE 2008)*. ACM, New York, NY, USA, 211–220.

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 *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys 2006)*. ACM, New York, NY, USA, 73–85.

Thomas Ball and Jakub Daniel. 2015. Deconstructing Dynamic Symbolic Execution, In Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering. (2015).

Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic Detection of Floating-point Exceptions. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013)*. ACM, New York, NY, USA, 549–560.

Clark Barrett and Cesare Tinelli. 2007. CVC3. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*. Springer-Verlag, Berlin, Heidelberg, 298–302.

Tom Bergan, Dan Grossman, and Luis Ceze. 2014. Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014)*. ACM, New York, NY, USA, 491–506.

Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh (Eds.). 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press. 980 pages.

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

Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT: a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*. ACM, New York, NY, USA, 234–245.

David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. *Botnet Detection: Countering the Largest Security Threat*. Springer US, Chapter Automatically Identifying Trigger-based Behavior in Malware, 65–88.

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*. Springer-Verlag, Berlin, Heidelberg, 463–469.

Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-world Software Testing. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys 2011)*. 183–198.

Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*. IEEE Computer Society, Washington, DC, USA, 443–446.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008)*. 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 *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*. ACM, New York, NY, USA, 322–335.

Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proceedings of the 33rd Inter. Conf. on Software Engineering (ICSE 2011)*. ACM, 1066–1071.

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

Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song. 2013. HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism. In *18th European Symposium on Research in Computer Security (ESORICS 2013)*. 164–181.

Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP 2012)*. IEEE Computer Society, Washington, DC, USA, 380–394.

Avik Chaudhuri and Jeffrey S. Foster. 2010. Symbolic Security Analysis of Ruby-on-rails Web Applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*. ACM, New York, NY, USA, 585–594.

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

Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic Crosschecking of Floating-point and SIMD Code. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys 2011)*. ACM, New York, NY, USA, 315–328.

Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. 2014. Symbolic Crosschecking of Data-Parallel Floating-Point Code. *IEEE Transactions on Software Engineering* 40, 7 (2014), 710–737.

Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proceedings of the 23rd USENIX Security Symposium*. 95–110.

Christoph Csallner and Yannis Smaragdakis. 2005. Check 'N' Crash: Combining Static Checking and Testing. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*. ACM, New York, NY, USA, 422–431.

Drew Davidson, Benjamin Moench, Somesh Jha, and Thomas Ristenpart. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Conference on Security (SEC 2013)*. USENIX Association, Berkeley, CA, USA, 463–478.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

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

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

Celina Gomes do Val. 2014. *Conflict-Driven Symbolic Execution: How to Learn to Get Better*. MSc Thesis. University of British Columbia.

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

Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. 2009. Precise Pointer Reasoning for Dynamic Test Generation. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM, New York, NY, USA, 129–140.

Dawson Engler and Daniel Dunbar. 2007. Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA 2007)*. ACM, New York, NY, USA, 1–4.

Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. 2013. Con2Colic Testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 37–47.

Vijay Ganesh. 2007. *Decision Procedures for Bit-vectors, Arrays and Integers*. Ph.D. Dissertation. https://ece.uwaterloo.ca/~vganesh/Publications_files/vg2007-PhD-STANFORD.pdf.

Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*. Springer-Verlag, Berlin, Heidelberg, 519–531.

Jaco Geldenhuys, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. 2013. Bounded Lazy Initialization. In *5th International NASA Formal Methods Symposium (NFM 2013)*, Guillaume Brat, Neha Rungta, and Arnaud Venet (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 229–243.

Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*. ACM, 47–54.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*. ACM, New York, NY, USA, 213–223.

Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages.

Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2008)*.

Patrice Godefroid and Daniel Luchaup. 2011. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011)*. ACM, New York, NY, USA, 23–33.

Alex Groce and Willem Visser. 2002. Model Checking Java Programs Using Structural Heuristics. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*. ACM, New York, NY, USA, 12–21.

Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. 2015. Assertion Guided Symbolic Execution of Multithreaded Programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 854–865.

Nguyen Minh Hai, Mizuhito Ogawa, and Quan Thanh Tho. 2016. *Obfuscation Code Localization Based on CFG Generation of Malware*. Springer International Publishing, Cham, 229–247.

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

Martin Hentschel, Richard Bubel, and Reiner Hähnle. 2014. Symbolic Execution Debugger (SED). In *Proceedings of Runtime Verification 2014 (RV 2014)*. Springer, 255–262.

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

Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. 2012. *SymDroid: Symbolic Execution for Dalvik Bytecode*. Technical Report CS-TR-5022. Depart. of Computer Science, Univ. of Maryland, College Park.

Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 177–187.

Paul A. Karger and Roger R. Schell. 1974. *Multics security evaluation: Vulnerability analysis*. Technical Report. HQ Electronic Systems Division: Hanscom AFB, MA.

Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. 2010. Mixing Type Checking and Symbolic Execution. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010)*. ACM, New York, NY, USA, 436–447.

Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*. Springer-Verlag, Berlin, Heidelberg, 553–568.

James C. King. 1975. A New Approach to Program Testing. In *Proceedings of the International Conference on Reliable Software*. ACM, New York, NY, USA, 228–233.

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

Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*. ACM, New York, NY, USA, 193–204.

Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 449–459.

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

Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. 2016. JDart: A Dynamic Symbolic Analysis Framework. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016)*. Springer-Verlag New York, Inc., New York, NY, USA, 442–459.

Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *Proceedings of the 29th Intern. Conf. on Software Engineering (ICSE 2007)*. IEEE Computer Society, Washington, DC, USA, 416–426.

Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 337–348.

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

Jonathan Pincus and Brandon Baker. 2004. Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security Privacy* 2, 4 (July 2004), 20–27.

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

Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*. ACM, New York, NY, USA, 179–180.

Corina S. Păsăreanu, Neha Rungta, and Willem Visser. 2011. Symbolic Execution with Mixed Concrete-symbolic Solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011)*. ACM, New York, NY, USA, 34–44.

Corina S. Păsăreanu and Willem Visser. 2004. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In *Model Checking Software: 11th International SPIN Workshop (SPIN 2004)*. Springer Berlin Heidelberg, 164–181.

Jaideep Ramachandran, Corina Păsăreanu, and Thomas Wahl. 2015. Symbolic Execution for Checking the Accuracy of Floating-Point Programs. *ACM SIGSOFT Softw. Engineering Notes* 40, 1 (Feb. 2015), 1–5.

David A. Ramos and Dawson Engler. 2015. Under-constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC 2015)*. USENIX Association, Berkeley, CA, USA, 49–64.

Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32nd ACM/IEEE Intern. Conf. on Software Engineering (ICSE 2010)*. ACM, New York, NY, USA, 445–454.

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* 41, 7 (July 2015), 639–660.

Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 31–54.

Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended Symbolic Execution on Binary Programs. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM, New York, NY, USA, 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 *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP 2010)*. IEEE Computer Society, Washington, DC, USA, 317–331.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Conf. on Security (SEC 2011)*. USENIX Association, 25–25.

Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*. Springer-Verlag, Berlin, Heidelberg, 419–423.

Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 488–498.

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

Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. 2007. Abstracting Symbolic Execution with String Analysis. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. IEEE Computer Society, Washington, DC, USA, 13–22.

Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2008)*.

Asankhaya Sharma. 2014. Exploiting Undefined Behaviors for Efficient Symbolic Execution. In *Companion Proceedings of the 36th Intern. Conf. on Software Engineering (ICSE Companion 2014)*. ACM, 727–729.

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 Symposium (NDSS 2015)*.

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

Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The Concurrency Intermediate Verification Language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2015)*. ACM, New York, NY, USA, Article 61, 12 pages.

Jiri Slaby, Jan Strejcek, and Marek Trtík. 2013. Compact Symbolic Execution. In *11th International Symposium on Automated Technology for Verification and Analysis (ATVA 2013)*. 193–207.

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 *Proceedings of the 4th International Conference on Information Systems Security (ICISS 2008)*. Springer-Verlag, Berlin, Heidelberg, 1–25.

Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu. 2011. CORAL: Solving Complex Constraints for Symbolic Pathfinder. In *Proceedings of the Third International NASA Formal Methods Symposium (NFM 2011)*. Springer-Verlag, Berlin, Heidelberg, 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 2016)*.

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 *Proceedings of the 22nd Intern. Conf. on Computer Aided Verification (CAV 2010)*. Springer-Verlag, Berlin, Heidelberg, 288–305.

Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: White Box Test Generation for .NET. In *Proceedings of the 2nd Intern. Conf. on Tests and Proofs (TAP 2008)*. Springer-Verlag, Berlin, Heidelberg, 134–153.

Marek Trtík and Jan Strejček. 2014. *Symbolic Memory with Pointers*. Springer International Publishing, Cham, 380–395.

Sharath K. Udupa, Saumya K. Debray, and Matias Madou. 2005. Deobfuscation: Reverse Engineering Obfuscated Code. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005)*. IEEE Computer Society, Washington, DC, USA, 45–54.

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012)*. ACM, New York, NY, USA, Article 58, 11 pages.

Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*. ACM, New York, NY, USA, 97–107.

Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. 2006. Test Input Generation for Java Containers Using State Matching. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA 2006)*. ACM, New York, NY, USA, 37–48.

Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. 2011. *Linear Obfuscation to Combat Symbolic Execution*. Springer Berlin Heidelberg, Berlin, Heidelberg, 210–226.

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

Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*. 359–368.

Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*. ACM, New York, NY, USA, 351–363.

Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Proceedings of the 22nd ACM SIGSAC Conf. on Computer and Communications Security (CCS 2015)*. ACM, 732–744.

Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP 2015)*. IEEE Computer Society, Washington, DC, USA, 674–691.

Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 144–154.

Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *21st Annual Network and Distributed System Security Symposium (NDSS 2014)*.

Kim Zitter. 2013. How a Crypto Backdoor Pitted the Tech World Against the NSA. (2013). https://www.wired.com/2013/09/nsa-backdoor/all/.