Fuzzing Symbolic Expressions

Luca Borzacchiello
DIAG Department
Sapienza University of Rome
Rome, Italy
borzacchiello@diag.uniroma1.it

Emilio Coppa
DIAG Department
Sapienza University of Rome
Rome, Italy
coppa@diag.uniroma1.it

Camil Demetrescu
DIAG Department
Sapienza University of Rome
Rome, Italy
demetres@diag.uniroma1.it

***Abstract*—Recent years have witnessed a wide array of results in software testing, exploring different approaches and methodologies ranging from fuzzers to symbolic engines, with a full spectrum of instances in between such as concolic execution and hybrid fuzzing. A key ingredient of many of these tools is Satisfiability Modulo Theories (SMT) solvers, which are used to reason over symbolic expressions collected during the analysis. In this paper, we investigate whether techniques borrowed from the fuzzing domain can be applied to check whether symbolic formulas are satisfiable in the context of concolic and hybrid fuzzing engines, providing a viable alternative to classic SMT solving techniques. We devise a new approximate solver,** FUZZY-SAT**, and show that it is both competitive with and complementary to state-of-the-art solvers such as Z3 with respect to handling queries generated by hybrid fuzzers.**

***Index Terms*—concolic execution, fuzzing testing, SMT solver**

I. INTRODUCTION

The automatic analysis of modern software, seeking for high coverage and bug detection is a complex endeavor. Two popular approaches have been widely explored in the literature: on one end of the spectrum, *coverage-guided fuzzing* starts from an input seed and applies simple transformations (mutations) to the input, re-executing the program to be analyzed to increase the portion of explored code. The approach works particularly well when the process is guided and informed by code coverage, with a nearly-native execution time per explored path [1], [2]. On the other end of the spectrum, *symbolic execution* (SE) assigns symbolic values to input bytes and builds expressions that describe how the program manipulates them, resorting to satisfiability modulo theories (SMT) [3] solver queries to reason over the program, e.g., looking for bug conditions. A popular variant of SE is *concolic execution* (CE), which concretely runs one path at a time akin to a fuzzer, collecting branch conditions along the way [4], [5]. By systematically negating these conditions, it steers the analysis to take different paths, aiming to increase code coverage. The time per executed path is higher than fuzzing but the aid of a solver allows for a smaller number of runs.

Different ideas have been proposed to improve the effectiveness of analysis tools by combining ideas from both fuzzing and SE somewhere in the middle of the spectrum. As a prominent example, *hybrid fuzzing* couples a fuzzer with a symbolic executor to enable the exploration of complex branches [4], [6]. Compared to base fuzzing, this idea adds a heavy burden due to the lack of scalability of symbolic execution. It is therefore of paramount importance to speed up the symbolic part of the exploration.

While there is no clear winner in the software testing spectrum, tools that hinge upon an SMT solver get a high price to pay in terms of running time, limiting their throughput.

Contributions. As a main contribution, this paper addresses the following research question:

> *can we avoid using accurate but costly SMT solvers, replacing them with an approximate solver to test satisfiability in the context of software testing?*

We attempt to positively answer this question, devising FUZZY-SAT, an approximate solver that borrows ideas from the fuzzing domain. Our solver is tailored to the symbolic expressions generated by concolic engines and can replace classic SMT solvers in this context. By analyzing the expressions contained in symbolic queries, FUZZY-SAT performs *informed* mutations to possibly generate new valuable inputs. To demonstrate the potential behind FUZZY-SAT, we present FUZZOLIC, a new hybrid fuzzer based on QEMU. To show that FUZZY-SAT can be used in other frameworks, we integrate it also in QSYM [4]. In our experimental evaluation:

1) we compare FUZZY-SAT to the SMT solver Z3 [7] and the approximate solver JFS [8] on queries issued by QSYM, which we use as a mature baseline. Our results suggest that FUZZY-SAT can provide a nice tradeoff between speed and solving effectiveness, i.e., the number of queries found satisfiable by a solver.
2) we show that FUZZY-SAT allows QSYM to find more bugs on the LAVA-M dataset [9] compared to Z3.
3) we evaluate FUZZOLIC on 12 real-world programs against state-of-the-art fuzzers including AFL++ [2], ECLIPSER [10], and QSYM, showing that it can reach higher code coverage than the competitors.

To facilitate extensions of our approach, we make our contributions available at:

https://season-lab.github.io/fuzzolic/

II. BACKGROUND

FUZZY-SAT takes inspiration from two popular software testing techniques [11]: symbolic execution [12] and coverage-

This paper is supported in part by European Union's Horizon 2020 research and innovation programme (grant agreement No. 830892, project SPARTA).

based grey-box fuzzing [13]. We now review the inner-workings of these two approaches, focusing on recent works that are tightly related to the ideas explored in this paper.

Symbolic execution. The key idea behind this technique is to execute a program over *symbolic*, rather than *concrete*, inputs. Each symbolic input can, for instance, represent a byte read by the program under test from an input file and initially evaluate to any value admissible for its data type (e.g., $[0, 255]$ for an unsigned byte). SE builds expressions to describe how the program manipulates the symbolic inputs, resorting to SMT solver queries to reason over the program state. In particular, when a branch condition b is met during the exploration, SE checks using the solver whether both directions can be taken by the program for some values of the inputs, forking the execution state in case of a positive answer. When forking, SE updates the list of *path constraints* π that must hold true in each state: b is added in the state for the *true* branch, while $\neg b$ is added to the state for the *false* branch. At any time, the symbolic executor can generate concrete inputs, able to reproduce the program execution related to one state, by asking the solver an assignment for the inputs given π.

SE can be performed on binary code (e.g., ANGR [14], S^2E [15]) or on more high-level representations of a program (e.g., LLVM IR in KLEE [16], Java bytecode in SPF [17]). Besides software testing, SE has been extensively used during the last decade in the context of cybersecurity [18]–[20].

Concolic execution. A twist of SE designed with scalability in mind is concolic execution [21], which given a concrete input i, analyzes symbolically only the execution path taken by the program when running over i. To generate new inputs, the concolic executor can query an SMT solver using $\neg b \wedge \pi$, where b is a branch condition taken by the program in the current path while π is the set of constraints from the branches previously met along the path. A benefit from this approach is that the concolic executor only needs to query the solver for one of the two branch directions, as the other one is taken by the path under analysis. Additionally, if the program is actually executed concretely in parallel during the analysis, the concolic engine can at any time trade accuracy for scalability, by concretizing some of the input bytes and make progress in the execution using the concrete state. For instance, when analyzing a complex library function, the concolic engine may *concretize* the arguments for the function and execute it concretely, without issuing any query or making π more complex due to the library code but possibly giving up on some alternative inputs due to the performed concretizations.

A downside of most concolic executors is that they restart from scratch for each input driving the exploration, thus repeating analysis work across different runs. To mitigate this problem, QSYM [4] has proposed a concolic executor built through dynamic binary instrumentation (DBI) that cuts down the time spent for running the program by maintaining only the symbolic state and offloads completely the concrete state to the native CPU. Additionally, it simplifies the symbolic state by concretizing symbolic addresses [22], [23] but also generates inputs that can lead the program to access alternative memory locations. More recently, SYMCC [5] has improved the design of QSYM by proposing a source-based instrumentation approach that further reduces the emulation time.

Approximate constraint solving. Many queries generated by concolic executors are either unsatisfiable or cannot be solved within a limited amount of time [4]. This often is due to the complex constraints contained in π, which can impact the reasoning time even when the negated branch condition is quite simple. For this reason, QSYM has introduced *optimistic solving* that, in case of failure over $\neg b \wedge \pi$ due to unsatness or solving timeout, submits to the solver an additional query containing only $\neg b$: by patching the input i (used to drive the exploration) in a way that makes $\neg b$ satisfied, the executor is often able to generate valuable inputs for a program.

A different direction is instead taken by JFS [8], which builds on the experimental observation that SMT solvers can struggle on queries that involve floating-point values. JFS thus proposes to turn the query into a program, which is then analyzed using coverage-based grey-box fuzzing. More precisely, the constructed program has a point that is reachable if and only if the program's input satisfies the original query. The authors show that JFS is quite effective on symbolic expressions involving floating-point values but it struggles on integer values when compared to traditional SMT solvers.

Two very recent works, PANGOLIN [24] and TRIDENT [25], devise techniques to reduce the solving time in CE. PANGOLIN transforms constraints into a *polyhedral path abstraction*, modeling the solution space as a polyhedron and using, e.g., sampling to find assignments. TRIDENT instead exploits interval analysis to reduce the solution space in the SMT solver. Their implementations have not been released yet.

Coverage-based grey-box fuzzing. An orthogonal approach to SE is coverage-based grey-box fuzzing (CGF). Given an input queue q (initialized with some input seeds) and a program p, CGF picks an input i from q, randomly mutates some of its bytes to generate i' and then runs p over i': if new code is executed (covered) by p compared to previous runs on other inputs, then CGF deems the input *interesting* and adds it to q. This process is then repeated endlessly, looking for crashes and other errors during the program executions.

American Fuzzy Lop (AFL) [1] is the most prominent example of CGF. To track the coverage, it can dynamically instrument at runtime a binary or add source-based instrumentation at compilation time. The fuzzing process for each input is split into two main stages. In the first one, AFL scans the input and *deterministically* applies for each position a set of mutations, testing the effect of each mutation on the program execution in isolation. In the second stage, AFL instead performs several mutations in sequence, i.e., *stacking* them, over the input, *non deterministically* choosing which mutations to apply and at which positions. The mutations in the two stages involve simple and fast to apply transformations such as flipping bits, adding or subtracting constants, removing bytes, combining different inputs, and several others [1].

Hybrid fuzzing. Although CGF fuzzers have found thou-

sands of bugs in the last years [26], [27], there are still scenarios where their mutation strategy is not effective. For instance, they may struggle on checks against magic numbers, whose value is unlikely to be generated with random mutations. As these checks may appear early in the execution, fuzzers may soon *get stuck* and stop producing interesting inputs. For this reason, a few works have explored combinations of fuzzing with symbolic execution, proposing *hybrid fuzzing*. DRILLER [6] alternates AFL and ANGR, temporarily switching to the latter when the former is unable to generate new interesting inputs for a specific budget of time. QSYM proposes instead to run a concolic executor in parallel with AFL, allowing the two components to share their input queues and continuously benefit from the work done by each other.

Recent improvements in coverage-guided fuzzing. During the last years, a large body of works has extended CGF, trying to make it more effective without resorting to heavyweight analyses such as symbolic execution. LAF-INTEL [28] splits multi-byte checks into single-byte comparisons, helping the fuzzer track the intermediate progress when reasoning on a branch condition. VUZZER [29] integrates dynamic taint analysis (DTA) [30] into the fuzzer to identify which bytes influence the operands in a branch condition, allowing it to bypass, e.g., checks on magic numbers. ANGORA [31] further improves this idea by performing multi-byte DTA and using gradient descent to effectively mutate the tainted input bytes.

As DTA can still put a high burden on the fuzzing strategy, some works have recently explored lightweight approximate analyses that can replace it. REDQUEEN [32] introduces the concept of *input-to-state correspondence*, which captures the idea that input bytes often flow directly, or after a few simple encodings (e.g., byte swapping), into comparison operands during the program execution. To detect this kind of input dependency, REDQUEEN uses *colorization* that inserts random bytes into the input and then checks whether some of these bytes appear, as is or after few simple transformations, in the comparison operands when running the program. Input-to-state relations can be exploited to devise effective mutations and bypass several kinds of validation checks.

WEIZZ [33] explores instead a different approach that flips one bit at a time on the entire input, checking after each bit flip which comparison operands have changed during the program execution, possibly suggesting a dependency between the altered bit and the affected branch conditions. While more accurate than colorization, this approach may incur a large overhead, especially in presence of large inputs. Nonetheless, WEIZZ is willing to pay this price as the technique allows it to also heuristically locate fields and chunks within an input, supporting *smart mutations* [26] to effectively fuzz applications processing structured input formats.

SLF [34] exploits a bit flipping strategy similar to WEIZZ to generate valid inputs for an application even when no meaningful seeds are initially available for it. Thanks to the input dependency analysis, SLF can identify fields into the input and then resort to a gradient-based multi-goal search heuristic to deal with interdependent checks in the program.

ECLIPSER [10] identifies a dependency between an input byte i_k and a branch condition b whenever the program decision on b is affected when running the program on inputs containing different values for i_k. ECLIPSER builds approximate path constraints by modeling each branch condition met along the program execution as an interval. In particular, given a branch b, it generates a new input using a strategy similar to concolic execution, by looking for input values that satisfy the interval from $\neg b$ as well as any other interval from previous branches met along a path. To find input assignments, ECLIPSER does not use an SMT solver but resorts to lightweight techniques that work well in presence of intervals generated by linear or monotonic functions.

III. APPROACH

Recent coverage-guided fuzzers perform input mutations based of a knowledge on the program behavior that goes beyond the simple code coverage. Concolic executors by design build an accurate description of the program behavior, i.e., symbolic expressions, but outsource completely the reasoning to a powerful but expensive SMT solver, which is typically treated as a black box. In this paper, we explore the idea that a concolic executor can learn from the symbolic expressions that it has built and use the acquired knowledge to apply simple but fast input transformations, possibly solving queries without resorting to an SMT solver. The key insight is that given a query $\neg b \wedge \pi$, the input i that has driven the concolic exploration satisfies by design π. Hence, we propose to build using input mutations a new test case i' that satisfies $\neg b$ and is similar enough to i so that π remains satisfied by i'. In the remainder of this section, we present the design of FUZZY-SAT, an approximate solver that explores this direction by borrowing ideas from the fuzzing domain to efficiently solve queries generated by concolic execution.

A. Reasoning primitives for concolic execution

While SMT solvers typically offer a rich set of solving primitives, enabling reasoning on formulas generated from quite different application contexts, concolic executors such as QSYM are instead built on top of a few but essential primitives. In this paper, we focus on these primitives without claiming that FUZZY-SAT can replace a full-fledged SMT solver in a general context. FUZZY-SAT exposes the following primitives:

- SOLVE(e, π, i, opt): returns an assignment for the symbolic inputs in $e \wedge \pi$ such that the expression $e \wedge \pi$ is satisfiable. The flag *opt* indicates whether optimistic solving should be performed in case of failure. This primitive is used by concolic engines when negating a branch condition b, hence $e = \neg b$.
- SOLVEMAX(e, π, i) (resp. SOLVEMIN(e, π, i)): returns an assignment that maximizes (resp. minimizes) e while making π satisfiable. Concolic executors use these primitives before concretizing a symbolic memory address e to keep the exploration scalable. These functions are thus

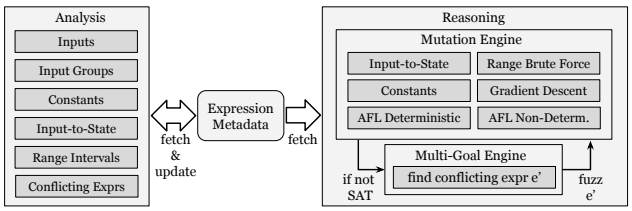


Fig. 1. Internal architecture of FUZZY-SAT.

used to generate alternative inputs that steer the program to read/write at boundary addresses.

- SOLVEALL(e, π, i): combines SOLVEMIN and SOLVEMAX, yielding intermediate assignments identified during the reasoning process as well. This primitive is valuable in the presence of symbolic memory addresses accessing a jump table or when the instruction pointer becomes symbolic during the exploration.

Two main aspects differentiate these primitives in FUZZY-SAT with respect to their counterpart from an SMT solver.

First, FUZZY-SAT is an approximate solver and thus it cannot guarantee that no valid assignment exists in case of failure of SOLVE, i.e., FUZZY-SAT cannot prove that an expression $e \wedge \pi$ is unsatisfiable. Similarly, given an expression e, FUZZY-SAT may fail to find its global minimum/maximum value or to enumerate assignments for all its possible values.

Another crucial difference is that FUZZY-SAT requires that the concolic engine provides the input test case i that was used to steer the symbolic exploration of the program under test. This is essential as FUZZY-SAT builds assignments by mutating the test case i based on facts that are learned when analyzing e and π. Given an assignment a returned by FUZZY-SAT, a new input test case i' can be built by patching the bytes in i that are assigned by a.

B. Overview

Architecture. To support the primitives presented in Section III-A, the architecture of FUZZY-SAT (Figure 1) has been structured around three main building blocks: the *analysis stage*, the *expression metadata*, and the *reasoning stage*.

The analysis stage (§ III-C) is designed to analyze symbolic expressions, extracting valuable knowledge to use during the reasoning stage. It starts by identifying which input bytes i_k from the input i are involved in an expression and how they are grouped. It detects input-to-state relations (§ II) and collects constants appearing in the expression for later use in the mutation phase. Expressions that constrain the interval of admissible values for a set of inputs, dubbed *range constraints*, such as $i_0 < 10$, are identified to keep track of the *range intervals* over the symbolic inputs. Finally, this stage detects whether the current expression shares input bytes with other expressions previously processed by the analysis component, possibly pinpointing *conflicts* that may result when mutating these bytes during the reasoning stage.

The expression metadata maintains the knowledge of FUZZY-SAT on the expressions processed by the analysis stage over time. Internally, it is implemented as a set of data structures optimized for fast lookup of different kinds of properties related to an expression (and its subexpressions). It is updated by the analysis stage and queried by both stages.

Finally, the reasoning stage is where FUZZY-SAT exploits the knowledge over the expressions to effectively fuzz the input test case and possibly generate valid assignments. To reach this goal, a mutation engine (§ III-D) is used to perform a set of transformations over the input bytes involved in an expression e looking for an assignment that satisfies e and π (SOLVE) or maximizes/minimizes e while satisfying π (other primitives). When this step finds assignments for e, but none of them satisfies π, then FUZZY-SAT performs a multi-goal strategy, which is not limited to changing the input bytes involved in e, but attempts to alter other input bytes that are involved in conflicting expressions present in π.

```
function SOLVE(e, π, i, opt):
1   M ← ANALYZE(π, M)
2   M ← ANALYZE(e, M)
3   a, SA ← MUTATE(e, π, i, M)
4   if a is not NULL then return a
5   a ← PICKBESTASSIGNMENT(π, SA)
6   if a is not NULL then
7       M′ ← FIXINPUTBYTES(a, M)
8       CC ← GETCONFLICTINGEXPRESSIONS(e, π, M′)
9       for e′ ∈ CC do
10          a′, SA ← MUTATE(e′, π, i, M′)
11          if a′ is not NULL then return a′
12          a′ ← PICKBESTASSIGNMENT(π, SA)
13          if a′ is NULL then break
14          a ← a′
15          M′ ← FIXINPUTBYTES(a, M′)
16  if opt then
17      if a is NULL then  a ← MUTATEOPT(e, π, i, M)
18      return a
19  return NULL
```

Algorithm 1: SOLVE implementation of FUZZY-SAT: analysis stage in light gray, reasoning stage in dark gray (initial mutations due to e at lines 3-4, multi-goal strategy at lines 5-15, and optimistic solving at lines 16-18).

Implementing the reasoning primitives. Algorithm 1 shows the interplay of these three components in FUZZY-SAT when considering the primitive SOLVE. Lines 1 and 2 execute the analysis stage by invoking the ANALYZE function on π and e, respectively. ANALYZE updates the expression metadata M, adding any information that could be valuable during the reasoning stage. Since concolic engines would typically call SOLVE several times during the symbolic exploration, providing each time a π that is the conjunction of branch conditions met along the path and which have been already analyzed by FUZZY-SAT in previous runs of SOLVE, the call at line 1 does not lead FUZZY-SAT to perform any work in most scenarios as the expression metadata M already has a cache containing knowledge about expressions in π.

Lines 3-18 instead comprise the reasoning stage and can be divided into three main phases. First, the MUTATE function is called at line 3 to run the mutation engine, restricting the transformations on input bytes that are involved in the expression e. When MUTATE finds an assignment a that

satisfies both e and π, SOLVE returns it at line 4 without any further work. On the other hand, when a is invalid but some assignments SA found by MUTATE make at least e satisfiable, then SOLVE starts the multi-goal phase (lines 5-15). To this end, FUZZY-SAT uses function PICKBESTASSIGNMENT to select the best candidate assignment a from SA[1] and then *fixes* the input bytes assigned by a using function FIXINPUTBYTES to prevent further calls of MUTATE from altering these bytes. It then reruns the mutation engine considering an expression e' which has been marked as in conflict with e during the analysis stage. This process is repeated as long as three conditions hold: (a) $e \wedge \pi$ is not satisfied (line 11), (b) MUTATE returns at least one assignment in SA for e' (line 13), and (c) there is still a conflicting expression left to consider (condition at line 9).

The multi-goal strategy in FUZZY-SAT employs a greedy approach without ever performing backtrack (e.g., reverting the effects of FIXINPUTBYTES in case of failure) as it trades accuracy for scalability. Indeed, FUZZY-SAT builds on the intuition that by altering a *few* bytes from the input test case i, it is possible in several cases to generate valid assignments. Additionally, since many queries generated by a concolic engine are unsatisfiable, increasing the complexity of this strategy would impose a large burden on FUZZY-SAT.

The last phase of SOLVE (lines 16-18) has been devised to support optimistic solving in FUZZY-SAT. When the Boolean opt is true, FUZZY-SAT returns the last candidate assignment found by the mutation engine, which by design satisfies the expression e. However, since the previous calls to the mutation engine in SOLVE may have failed to find an assignment a for e due to the constraints resulting from the analysis of expressions from π, FUZZY-SAT as last resort uses a variant of the function MUTATE, called MUTATEOPT, that ignores these constraints and exploits only knowledge resulting from e when performing transformations over the input bytes.

The other reasoning primitives (SOLVEMIN, SOLVEMAX, and SOLVEALL, respectively) follow a workflow similar to SOLVE and we do not present their pseudocode due to lack of space. In the remainder of this section, we focus on the internal details of functions ANALYZE (§ III-C) and MUTATE (§ III-D), which are crucial core elements of FUZZY-SAT.

C. Analyzing symbolic expressions

We now present the details of the main analyses integrated into the ANALYZE function, which incrementally build the knowledge of FUZZY-SAT over an expression e.

Detecting inputs and input groups. The first analysis identifies which input bytes i_k are involved in an expression and evaluates how these bytes are grouped. In particular, FUZZY-SAT checks whether the expression can be regarded as an *input group*, i.e., the expression is equivalent to a concatenation ($+\!\!+$) of input bytes or constants that never *mix* their bits. Single byte expressions are also detected as input groups.

[1]We pick an a that maximizes the number of expressions satisfied in π.

Examples:

- expression $i_1 +\!\!+ i_0$ contains inputs i_0 and i_1, and it is an input group since the bits from these bytes do not mix with each other but are just appended;
- expression $0 +\!\!+ i_0$ contains input i_0 and it is a 1-byte input group as it is a zero-extend operation on i_0;
- expression $i_1 + i_0$ contains inputs i_0 and i_1, but it is not an input group as bits from i_0 are mixed, i.e., added, with bits from i_1;
- expression $(0 +\!\!+ i_0) + (i_1 \ll 8)$ contains inputs i_0 and i_1, and it is an input group as the expression is equivalent to $i_1 +\!\!+ i_0$, which is an input group.

Given an expression e, FUZZY-SAT stores in the expression metadata M the list of inputs involved in e, whether e is an input group, and the list of input groups *contained* in e when recursively considering subexpressions of e.

Detecting uniquely defined inputs. A crucial information about an input byte is knowing whether its value is fixed to a single value, dubbed *uniquely defined* in our terminology, due to one equality constraint that involves it. Indeed, it is not productive to fuzz input bytes whose value is fixed to a single constant. Given an expression e, then:

- if e is an equality constraint and one of its operands is an input group, or contains exactly only one input group, while the other operand is a constant, then M is updated to reflect that the bytes in the group will be uniquely defined due to e if e is later added to π;
- an input in e is marked as uniquely defined whenever a constraint from π marks it as uniquely defined;
- the input group in e (if any) is marked as uniquely defined whenever the inputs forming it are all uniquely defined due to constraints in π;

Example. The expression $i_1 +\!\!+ i_0 == 0\text{xABCD}$ makes FUZZY-SAT mark inputs i_0 and i_1 as uniquely defined. If this expression is later added to π, then i_0 and i_1 will be considered uniquely defined in other expressions, disabling fuzzing on their values.

Detecting input-to-state branch conditions. This analysis checks whether e contains at least one operand that has input-to-state correspondence (§ II). In FUZZY-SAT we use the following conditions to detect this kind of branch conditions: (a) e matches the pattern $e'\ op_{cmp}\ e''$, where op_{cmp} is a comparison operator (e.g., $\geq$, $==$, etc.) and (b) one operand (e' or e'') is an input group. When e is a Boolean negation, FUZZY-SAT recursively analyzes the subexpression.

Example. The expression $10 \geq i_1 +\!\!+ i_0$ is an input-to-state branch condition as $\geq$ is a comparison operator and $i_1 +\!\!+ i_0$ is an input group.

Detecting interesting constants. FUZZY-SAT checks the expression e, looking for constants that could be valuable during the reasoning stage, dynamically building a dictionary to

use during the transformations. When specific patterns are detected, FUZZY-SAT generates variants of the constants based on the semantics of the computation performed by e.

Example. When analyzing $i_1 \oplus 0\text{xF0} == 0\text{x0F}$, FUZZY-SAT collects the constants 0xF0, 0x0F, and 0xFF (i.e., $0\text{xF0} \oplus 0\text{x0F}$) since the computation is an exclusive or.

The patterns used to generate interesting constants can be seen as a relaxation of the concept of input-to-state relations.

Detecting range constraints. FUZZY-SAT checks whether e is a *range constraint*, i.e., a constraint that sets a lower bound or an upper bound on the values that are admissible for the input group in e (if any). For instance, FUZZY-SAT looks for constraints matching the pattern $e'\ op_{cmp}\ e''$ where e' is an input group, op_{cmp} is a comparison operator, and e'' is a constant value. Other equivalent patterns, such as $(e' - e'')\ op_{cmp}\ e'''$ where e' is an input group while e'' and e''' are constants, are detected as range constraints as well.

By considering bounds resulting from expressions in π and not only from e, FUZZY-SAT can compute refined range intervals for the input groups contained in an expression. To compactly and efficiently maintain these intervals, FUZZY-SAT uses *wrapped intervals* [35] which can transparently deal with both signed and unsigned comparison operators.

Examples.

- given the expressions $i_1 + i_0 > 10$ and $i_1 + i_0 \leq 30$, FUZZY-SAT computes the range interval $[11, 30]$ for the input group composed by i_0 and i_1;
- given the expression $(i_1 + i_0) + 0\text{xAAAA} <_{unsigned} 0\text{xBBBB}$, FUZZY-SAT computes the intervals $[0, 0\text{x1110}] \cup [0\text{x5556}, 0\text{xFFFF}]$ for i_0 and i_1, correctly modeling the wrap-around that may result in the two's complement representation.

Detecting conflicting expressions. The last analysis is devised to identify which expressions from π may *conflict* with e when assigning some of its input bytes. In particular, FUZZY-SAT marks an expression e' as in conflict with e whenever the set of input bytes in e' is not disjoint with the set from e.

Example. The expression $i_1 + i_0 > 10$ is in conflict with the expression $i_1 + i_2 < 20$ as they both contain the input byte i_1. Hence, fuzzing the first expression may negatively affect the second expression.

Computing the set of conflicting expressions is essential for performing the multi-goal strategy during the reasoning stage.

D. Fuzzing symbolic expressions

The core step during the reasoning stage of FUZZY-SAT is the execution of the function MUTATE, which attempts to find a valid assignment a. To reach this goal, MUTATE performs a sequence of mutations over the input test case i, returning as soon as a valid assignment is found by one of these transformations. When a mutation generates an assignment that satisfied e but not π, then MUTATE saves it into a set of candidate assignment SA, which could be valuable later on during the multi-goal strategy (§ III-B). In some cases, a transformation can determine that there exists a contradiction between e and the conditions in π, leading MUTATE to an early termination. Additionally, when MUTATE builds a candidate assignment a, it checks that a is consistent with the range intervals known for the modified bytes, discarding a in case of failure and avoiding the (possibly expensive) check over π. We now review in detail the input transformations performed by the function MUTATE.

Fuzzing input-to-state relations. When an expression e is an input-to-state branch condition (§ III-C), FUZZY-SAT tries to replace the value from one operand e' into the bytes composing the input group from the other (input-to-state) operand e''. If e' is not constant, then FUZZY-SAT gets its concrete value by evaluating e' on the test case i. When e' is constant and the relation is an equality, if the assignment does not satisfy π, then FUZZY-SAT deems the query unsatisfiable. Conversely, when the comparison operator is not an equality, FUZZY-SAT tests variants of the value from e', e.g., by adding or subtracting one to it, in the same spirit as done by REDQUEEN.

Example. Given $i_1 + i_0 == 0\text{xABCD}$, FUZZY-SAT builds the assignment $\{i_0 \leftarrow 0\text{xCD}, i_1 \leftarrow 0\text{xAB}\}$. If the range interval over i_0 is $[0\text{xDD}, 0\text{xFF}]$ due to constraints from π, then the assignment can be discarded without testing π, deeming the query unsatisfiable (but keeping the assignment in SA in case of optimistic solving).

Range interval brute force. When a range interval is known for an input group contained an expression e, FUZZY-SAT can use this information to perform brute force on its value and possibly find a valid assignment. In particular, when an expression contains a single input group and its range interval is less than 2048, FUZZY-SAT builds assignments that brute force all the possible values assignable to the group. If no valid assignment is found, then the query can be deemed unsatisfiable. If the interval is larger than 2048, then FUZZY-SAT only tests the minimum and maximum value of the interval. To make this input transformation less conservative, FUZZY-SAT runs it even when e contains at least one input group whose interval is less than[2] 512.

Example. Given the expression $(i_1 + i_0) * 0\text{xABCD} == 0\text{xCAFE}$ and the range interval $[1, 9]$ (built due to constraints from π) on the group g with i_0 and i_1, then FUZZY-SAT builds assignments for $g \in [1, 9]$, deeming the query unsatisfiable if none of them satisfies $e \wedge \pi$.

Trying interesting constants. For each constant c collected by ANALYZE when considering the expression e and for each input group g contained in e, FUZZY-SAT tries to set the bytes from g to the value c. Since constants are collected through *relaxed* patterns, FUZZY-SAT tests different encodings (e.g., little-endian, big-endian, zero-extension, etc.) for each constant to maximize the chances of finding a valid assignment.

[2]We pick the input group with the minimum range interval.

Example. Given the expression $(i_1 \mathbin{+\!\!+} i_0) * 100 == 200$ and assuming that ANALYZE has collected the constants $\{2, 99, 100, 101, 199, 200, 201\}$ where 2 was obtained as $200/100$, while other constants are obtained from 100 and 200, then FUZZY-SAT would find a valid assignment when testing $\{i_0 \leftarrow 2, i_1 \leftarrow 0\}$ ($c = 2$, little-endian encoding).

Gradient descent. Given an expression e, FUZZY-SAT tries to reduce the problem of finding a valid assignment for it to a minimization (or maximization) problem. This is valuable not only in the context of SOLVEMIN, SOLVEMAX, or SOLVEALL where this idea seems natural, but also when reasoning over the branch condition e in SOLVE. Indeed, any expression of the form $e' \; op_{cmp} \; e''$, where op_{cmp} is a comparison operator, can be transformed into an expression f amenable to minimization to find a valid assignment [31], e.g., $e' < e''$ can be transformed[3] into $f < 0$ with $f = e' - e''$.

The search algorithm implemented in FUZZY-SAT is inspired by ANGORA [31] and it is based on gradient descent. Although this iterative approach may fail to find a global minimum for f, a local minimum can be often *good enough* in the context of concolic execution as we do not always really need the global minimum but just an assignment that satisfies the condition, e.g., given $i_0 < 1$, the assignment $\{i_0 \leftarrow 0\text{x}0\}$ satisfies the condition even if the global minimum for $i_0 - 1$ is given by $\{i_0 \leftarrow 0\text{x}81\}$. For this reason, FUZZY-SAT in SOLVE can stop the gradient descent as soon an assignment satisfies both e and π. When the input groups from e have disjoint bytes, FUZZY-SAT computes the gradient considering groups of bytes, instead of computing it for each distinct byte, as this makes the descent more effective. In fact, reasoning on i_0 and i_1 as a single value is more appropriate when these bytes are used in a two-byte operation since gradient descent may fail when these bytes are considered independently.

Example. Given the expression $(i_0 \mathbin{+\!\!+} i_1) - 10 > (i_2 \mathbin{+\!\!+} i_3) - 5$ and a zero-filled input test case, then FUZZY-SAT transforms the expression into $((i_2 \mathbin{+\!\!+} i_3) - 5) - ((i_1 \mathbin{+\!\!+} i_0) - 10) < 0$, computes the gradients over the input groups $(i_1 \mathbin{+\!\!+} i_0)$ and $(i_2 \mathbin{+\!\!+} i_3)$, finding the assignment $\{i_0 \leftarrow 0\text{x}80, i_1 \leftarrow 0\text{x}06, i_2 \leftarrow 0\text{x}84, i_3 \leftarrow 0\text{x}01\}$ which makes the condition satisfied as $(0\text{x}80 \mathbin{+\!\!+} 0\text{x}06) - 10 = 32764 > -31748 = (0\text{x}84 \mathbin{+\!\!+} 0\text{x}01) - 5$.

Deterministic and non-deterministic mutations. These two sets of input transformations are inspired by the two mutation stages from AFL (§ II). Deterministic mutations include bit or byte flips, replacing bytes with interesting *well-known* constants (e.g., `MAX_INT`), adding or subtracting small constants from some input bytes. Non-deterministic mutations instead involve also transformations such as flipping of random bits inside randomly chosen input bytes, setting randomly chosen input bytes to randomly chosen interesting constants, subtracting or adding random values to randomly chosen bytes, and several others [1]. The main differences with respect to AFL are: (a) mutations are applied only on the input bytes involved in the expression e, (b) multi-byte mutations are considered only in the presence of multi-byte input groups, (c) for non-deterministic mutations, FUZZY-SAT generates k distinct assignments, with k equal to $\max\{100, n_i \cdot 20\}$ where n_i is the number of inputs involved in e, and for each assignment it applies a sequence (or stack) of n mutations ($n = 1 \ll (1 + rand(0, 7))$ as in AFL).

[3]For the sake of simplicity, we ignore in our examples the wrap-around.

E. Discussion

Similarly to fuzzers using dynamic taint analysis, FUZZY-SAT restricts mutations over the bytes that affect branch conditions during the program execution. However, it does not only understand *which* bytes influence the branch conditions but also reasons on *how* they affect them, possibly devising more effective mutations.

FUZZY-SAT shares traits with ANGORA, SLF, and ECLIPSER by integrating mutations based on gradient descent, a multi-goal strategy, and range intervals, respectively. Nevertheless, these techniques have been revisited and refined to work over symbolic constraints, which accurately describe the program state and are not available to these fuzzers.

FUZZY-SAT exposes primitives that are needed by concolic executors and that are typically offered by SMT solvers but it implements them in a fundamentally different way inspired by fuzzing techniques, trading accuracy for scalability.

Finally, FUZZY-SAT shares the same spirit of JFS but takes a rather different approach. While JFS builds a bridge between symbolic execution and fuzzers by turning expressions into a program to fuzz, FUZZY-SAT is designed to merge these two worlds, possibly devising *informed* mutations that are driven by the knowledge acquired by analyzing the expressions.

IV. IMPLEMENTATION

FUZZY-SAT is written in C (10K LoC) and evaluates queries in the language used by the Z3 Theorem Prover [7]. To efficiently evaluate an expression given a concrete assignment FUZZY-SAT uses a fork of Z3 where the `Z3_model_eval` function has been optimized to deal with full concrete models.

FUZZOLIC is a new concolic executor based on QEMU 4.0 (user-mode), written in C (20K LoC), that currently supports Linux x86_64 binaries. Its design overcomes one of the major problems affecting QSYM: FUZZOLIC decouples the tracer component, which builds the symbolic expressions, from the solving component, which reasons over them. This is required as recent releases of most DBI frameworks, such as PIN [36] on which QSYM is based on, do not allow an analysis tool to use external libraries (as the Z3 solver in case of QSYM) when they may produce side effects on the program under analysis [37]. This implementation constraint has made it very complex to port QSYM to newer releases of PIN, limiting its compatibility with recent software and hardware configurations[4]. To overcome this issue, the two components are executed into distinct processes in FUZZOLIC.

[4]QSYM has been recently removed from the Google project FuzzBench due to its instability on recent Linux releases [38].

In particular, the tracer runs under QEMU and generates symbolic expressions in a compact language, storing them into a shared memory that is also attached to the memory space of the solving component, which in turn submits queries to FUZZY-SAT to generate alternative inputs. Similarly to QSYM, FUZZOLIC runs in parallel with two coverage-guided fuzzers.

V. EVALUATION

In this section we address the following research questions:

- **RQ1**: How effective and efficient is FUZZY-SAT at solving queries generated by concolic executors?
- **RQ2**: How do different kinds of mutations help FUZZY-SAT in solving queries?
- **RQ3**: How does FUZZOLIC with FUZZY-SAT compare to state-of-the-art fuzzers on real-world programs?

Benchmarks. Throughout our evaluation, we consider the following 12 programs: advmng 2.00, bloaty rev `7c6fc`, bsdtar rev. `f3b1f`, djpeg v9d, jhead 3.00-5, libpng 1.6.37, lodepng-decode rev. `5a0dba`, objdump 2.34, optipng 0.7.6, readelf 2.34, tcpdump 4.9.3 (libpcap 1.9.1), and tiff2pdf 4.1.0. These targets have been heavily fuzzed by the community [27], and used in previous evaluations of state-of-the-art fuzzers [4], [5], [10], [32], [33]. As seeds, we use the AFL test cases [1], or when missing, minimal syntactically valid files [39].

Experimental setup. We ran our tests in a Docker container based on the Ubuntu 18.04 image, using a server with two Intel Xeon E5-4610v2@2.30 GHz CPUs and 256 GB of RAM.

A. RQ1: Solving effectiveness of FUZZY-SAT

To evaluate how effective and efficient is FUZZY-SAT at solving queries generated by concolic execution, we discuss an experimental comparison of FUZZY-SAT against the SMT solver Z3 and the approximate solver JFS. We first focus on SOLVE queries, collected by running the 12 programs under QSYM on their initial seed with optimistic solving disabled, comparing the solving time and the number of queries successfully proved as satisfiable when using these three solvers. Then, we analyze the performance of QSYM at finding bugs on the LAVA-M dataset [9] when using FUZZY-SAT with respect to when using Z3, implicitly considering the impact also of other reasoning primitives (e.g., SOLVEMAX) and from enabling optimistic solving in SOLVE. In these experiments, we consider QSYM instead of FUZZOLIC to avoid any bias resulting from its expression generation phase that could benefit FUZZY-SAT and impair the other solvers.

FUZZY-SAT vs Z3. Table I provides an overview of the comparison between FUZZY-SAT and Z3 on the queries generated when running the 12 benchmarks.

The first interesting insight is that only a small subset of the queries, i.e., less than 10%, has been proved satisfiable (even when considering together both solvers). The remaining queries are either proved unsatisfiable or make the solvers run out of the time budget (10 seconds for Z3, as in QSYM).

The second insight is that, when focusing on the queries that are satisfiable, FUZZY-SAT is able to solve the majority of them and can even perform better than Z3 on a few benchmarks: for instance, FUZZY-SAT solves 301 (236.7 + 64.3) queries on average on advmng, while Z3 stops at 243.7 (236.7 + 7). Although this may seem unexpected, this result is consistent with past evaluations from state-of-the-art fuzzers [10], [32] that have shown that a large number of branch conditions can be solved even without SMT solvers. Nonetheless, there are still a few queries were FUZZY-SAT is unable to find a valid assignment while Z3 is successful, e.g., FUZZY-SAT misses 7 queries on bloaty (but solves one query that makes Z3 run out of time). Assessing the impact of *solving* or *not solving* a query in concolic execution is a hard problem, especially when bringing into the picture hybrid fuzzing and its non-deterministic behavior. Hence, we only try to indirectly speculate on this impact by later discussing the results on the LAVA-M dataset and the experiments in Section V-C.

TABLE I
NUMBER OF QUERIES PROVED SATISFIABLE BY FUZZY-SAT W.R.T. Z3 (TIMEOUT 10 SECS). NUMBERS SHOW THE AVERAGE OF 5 RUNS. THE SPEEDUP CONSIDERS THE SOLVING TIME ON THE FULL SET OF QUERIES.

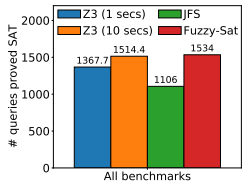
PROGRAM	# QUERIES	# QUERIES PROVED SAT BY			# SAT FUZZY-SAT DIV. BY # SAT Z3	SOLV. TIME SPEEDUP
		BOTH	Z3	FUZZY-SAT		
advmng	1481	236.7	+7.0	+64.3	1.24	17.1×
bloaty	2085	95.0	+7.0	+1.0	0.94	47.8×
bsdtar	325	124.0	+6.0	0	0.95	1.8×
djpeg	1245	189.0	+6.0	+11.0	1.03	34.3×
jhead	405	88.0	0	0	1.00	21.7×
libpng	1673	31.0	0	0	1.00	70.9×
lodepng	1531	100.3	+6.3	+4.7	0.98	75.6×
objdump	992	146.0	+4.0	0	0.97	30.6×
optipng	1740	42.0	0	0	1.00	67.3×
readelf	1055	150.0	+8.0	0	0.95	69.5×
tcpdump	409	58.3	+9.7	+28.7	1.28	37.3×
tiff2pdf	3084	164.0	+9.0	0	0.95	28.1×
G. MEAN	1335.4	118.7	+5.3	+9.1	1.02	31.2×

Lastly, we can see in Table I that on average FUZZY-SAT requires 31× less time than Z3 to reason over the queries from the 12 benchmarks. When putting together this result with the previous experimental insights, we could speculate why FUZZY-SAT could be beneficial in the context of concolic execution: it can significantly reduce the solving time during the concolic exploration while still be able to generate a large number of (possibly valuable) inputs.

One natural question is whether one could get the same benefits of FUZZY-SAT by drastically reducing the time budget given to Z3. To tackle this observation, Figure 2a reports the number of queries solved by Z3 when using a timeout of 1 second and Figure 2b shows how the speedup from FUZZY-SAT is reduced in this setup. FUZZY-SAT is still 9.5× faster than Z3 and the gap between the two in terms of solved queries increases significantly (+12% in FUZZY-SAT), suggesting that this setup of Z3 is not as effective as one may expect.

FUZZY-SAT vs JSF. One solver that shares the same spirit of FUZZY-SAT is JFS (§ II), which however is based on a different design. When considering the queries collected on the 12 benchmarks, it can be seen in Figure 2a that JFS is able to solve only 1106 queries, significantly less than the 1534 from FUZZY-SAT. On 127 out of the 325 queries from bsdtar, JFS has failed to generate the program to fuzz due

(a)



(b)

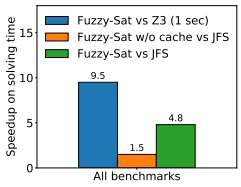


Fig. 2. FUZZY-SAT vs other solvers on the 12 benchmarks: (a) number of queries proved satisfiable and (b) speedup on the solving time.

TABLE II
BUGS FOUND ON LAVA-M IN 5H: AVG (MAX) NUMBER OVER 5 RUNS.

	base64	md5sum	uniq	who
QSYM WITH Z3	48 (48)	58 (58)	19 (29)	743 (795)
QSYM WITH FUZZY-SAT	48 (48)	61 (61)	19.7 (29)	2256.5 (2268)

to the large number of nested expressions contained in the queries, yielding a gap of 95 solved queries between two solvers. The remaining missed queries can be likely explained by considering that: (a) it is not currently possible to provide the input test case i used for generating the queries to the fuzzer executed by JFS [40], as JFS generates a program that takes an input that is different (in terms of size and structure) from i and builds its own set of seeds, (b) JFS does not provide specific insights to the fuzzer on how to mutate the input, and (c) JFS uses LIBFUZZER [41], which does not integrate several fuzzing techniques that have inspired FUZZY-SAT.

When considering the solving time, FUZZY-SAT is $1.5\times$ faster than JFS (Figure 2b). However, when enabling analysis cache in FUZZY-SAT, the speedup increases up to $4.8\times$.

JFS does not currently provide a C interface [42], requiring concolic executors to dump the queries on disk: as this operation can take a long time in presence of large queries, we do not consider JFS further in the other experiments.

FUZZY-SAT on LAVA-M. To test whether FUZZY-SAT can solve queries that are valuable for a concolic executor, we repeated the experiment on the LAVA-M dataset from the QSYM paper [4], looking for bugs within the four benchmarks base64, md5sum, uniq, and who. Table II reports the average and max number of bugs found during 5-hour experiments across 5 runs. QSYM with FUZZY-SAT finds on average more bugs than QSYM with Z3 on 3 out of 4 programs. In particular, the improvement is rather significant on who, where FUZZY-SAT allows QSYM to find $3\times$ more bugs compared to Z3, suggesting that trading performance for accuracy can be valuable in the context of hybrid fuzzing.

Interestingly, FUZZY-SAT was able to reveal bugs that the original authors from LAVA-M were unable to detect [9], e.g., FUZZY-SAT has revealed 136 new bugs on who. Since other works [10], [32] reported a similar experimental observation, the additional bugs are likely not false positives.

B. RQ2: Impact of different kinds of mutations in FUZZY-SAT

An interesting question is which mutations contribute at making FUZZY-SAT effective. Table III reports which transformations have been crucial to solve the queries from the 12 benchmarks, assigning a query to the multi-goal strategy when FUZZY-SAT had to reason over conflicting expressions from

TABLE III
EFFECTIVENESS OF THE DIFFERENT MUTATIONS FROM FUZZY-SAT: I2S (INPUT-TO-STATE), BF (R.I. BRUTE FORCE), IC (INTERESTING CONSTANTS), GD (GRADIENT DESCENT), D+ND (DETERMINISTIC AND NON-DETERMINISTIC MUTATIONS), MGS (MULTI-GOAL STRATEGY).

PROGRAM	I2S	BF	IC	GD	D+ND	MGS
advmng	176	31	74	2	18	0
bloaty	43	5	20	5	19	4
bsdtar	14	8	11	0	0	91
djpeg	98	29	28	6	14	25
jhead	27	5	41	6	8	0
libpng	14	8	7	1	1	0
lodepng	61	6	16	1	21	0
objdump	91	21	18	1	10	5
optipng	28	7	6	0	1	0
readelf	96	10	22	0	22	0
tcpdump	28	7	11	1	11	29
tiff2pdf	107	25	6	0	2	24
PERC. ON TOTAL	51.04%	10.56%	17.01%	1.50%	8.28%	11.60%

π to solve the query. FUZZY-SAT was able to solve more than 51% of the queries by applying input-to-state transformations, and an additional 17% was solved by exploiting the interesting constants collected during the analysis stage. Range interval brute-force was helpful on around 10% of the queries, while mutations inspired by AFL were beneficial in 8% of them. Gradient descent solved just 1.5% of the queries. However, two considerations must be taken into account: (a) the order of the mutations affect these numbers, as gradient descent is not used when previous (cheaper) mutations are successful, and (b) gradient descent is crucial for solving queries in SOLVEMIN, SOLVEMAX, and SOLVEALL, which are not considered in this experiment. Finally, the multi-goal strategy of FUZZY-SAT was essential for solving around 11% of the queries.

C. RQ3: FUZZY-SAT in FUZZOLIC

To further assess the effectiveness of FUZZY-SAT, we compare FUZZOLIC, which is built around this solver, against state-of-the-art binary open-source fuzzers on the 12 benchmarks, tracking the code coverage reached during 8-hour experiments (10 runs). Besides FUZZOLIC, we consider: (a) AFL++ [2] rev. `3f128` in QEMU mode, which integrates [43] the colorization technique from REDQUEEN, as well as other improvements to AFL proposed by the fuzzing community during the last years [44], (b) ECLIPSER rev. `b072f`, which devises one of the most effective *approximations* of concolic execution in literature, and (c) QSYM rev. `89a76` with Z3. As both FUZZOLIC and QSYM are hybrid fuzzers that are designed to run in parallel with two instances (F_m, F_s) of a coverage-guided fuzzer, we consider for a fair comparison AFL++ and ECLIPSER in a similar setup, running them in parallel to (F_m, F_s) and allowing the tools to sync their input queues [45]. Hence, each run takes $8 \times 3 = 24$ CPU hours. For F_m we use AFL++ in *master mode*, which performs deterministic mutations, while for F_s we use AFL++ in *slave mode* that only executes non-deterministic mutations. Since ECLIPSER does not support a parallel mode, we extended it to allow AFL++ to correctly pick inputs from its queue.

Figure 3 shows the code coverage reached by the different fuzzers on 8 out of 12 programs. On the remaining four

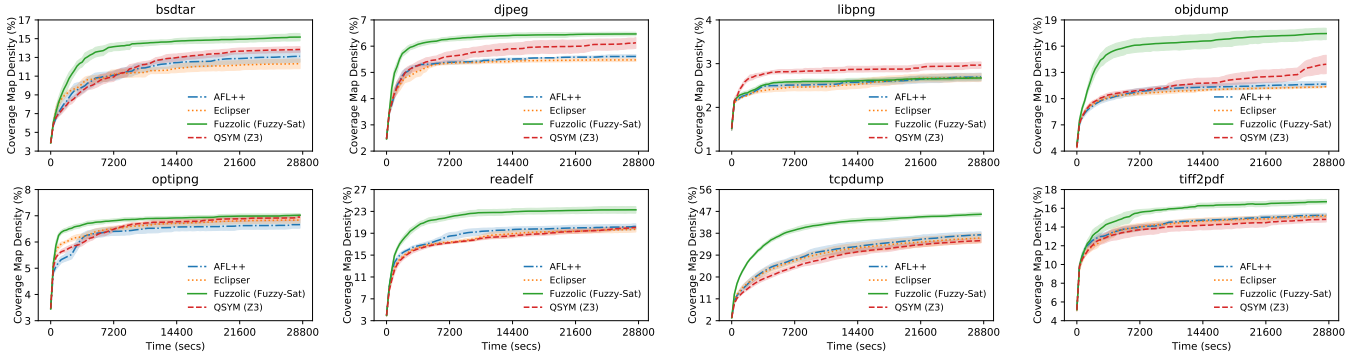


Fig. 3. Coverage map density reached by FUZZOLIC with FUZZY-SAT vs other state-of-the-art fuzzers. The shaded areas are the 95% confidence intervals.

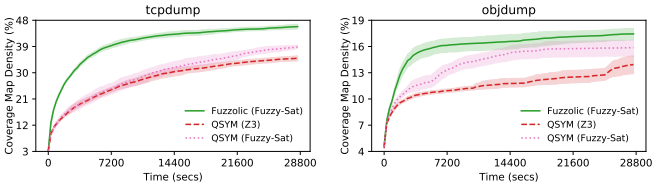


Fig. 4. Coverage map density: impact of FUZZY-SAT in QSYM.

benchmarks, the fuzzers reached soon a very similar coverage, making it hard to detect any significant trend and thus we omit their charts due to lack of space. Similar to other works [4], [5], we plot the density of the coverage map from F_s and depict the 95% confidence interval using a shaded area. As bitmap collisions may occur [46], we validated the trends by also computing the number of basic blocks [47].

FUZZOLIC reaches a higher code coverage than other solutions on 6 programs, i.e., bsdtar, djpeg, objdump, readelf, tcpdump, and tiff2pdf. In particular, tcpdump is the program where FUZZOLIC shines better, consistently showing over time an increase in the map density of 5% with respect to the second-best fuzzer (AFL++). On optipng, although FUZZOLIC appears to have an edge at the beginning of the experiment, it then reaches a coverage that is comparable to other fuzzers, which are all very close in performance. Finally, FUZZOLIC falls behind other approaches on libpng, pinpointing one case where FUZZY-SAT seems to underperform compared to Z3, as QSYM dominates on this benchmark.

When comparing closely FUZZOLIC to QSYM, the improvement in the coverage is likely due to the better scalability of the former with respect to the latter. For instance, on tcpdump FUZZOLIC performs concolic execution on 11089 inputs (1.8 secs/input), generating 14415 alternative inputs, while QSYM only analyzes 376 inputs (71.4 secs/input) and generates 786 alternative inputs. When considering libpng, FUZZOLIC is still faster than QSYM (8.8 secs/input vs 43.6 secs/input) but the number of inputs available in the queue from F_s (from which FUZZOLIC and QSYM pick inputs) over time is very low. Hence, the difference between FUZZOLIC and QSYM on libpng is due to a few but essential queries that Z3 is able to solve while FUZZY-SAT fails to reason on.

The better scalability of FUZZOLIC with respect to QSYM is given by the combination of an efficient solver (FUZZY-SAT) and an efficient tracer (FUZZOLIC). Indeed, when replacing Z3 with FUZZY-SAT in QSYM, this concolic executor improves its performance but still falls behind FUZZOLIC. Figure 4 compares the coverage of QSYM with the two solvers and FUZZOLIC on two benchmarks. On tcpdump, QSYM with FUZZY-SAT is able to analyze 2111 inputs (10.1 secs/input) and generate 3690 alternative inputs, improving the coverage by 4% on average with respect to Z3 but still performing worse than FUZZOLIC. Similarly, on objdump the improvement in QSYM due to FUZZY-SAT is even more noticeable but still QSYM cannot match the coverage reached by FUZZOLIC.

When comparing FUZZOLIC to ECLIPSER and AFL++, the results suggest that the integration of fuzzing techniques into a solver provides a positive impact. Indeed, while these fuzzers scales better than FUZZOLIC, processing hundreds of inputs per second, they lack the knowledge that FUZZY-SAT extracts from the symbolic expressions, which is used to perform effective mutations. Overall, colorization from AFL++ and approximate concolic execution from ECLIPSER seem to generate similar inputs on several benchmarks, yielding often a similar coverage in our parallel fuzzing setup. Moreover, despite FUZZOLIC may spend several seconds over a single input, it collects information that allows it to fuzz a large number of branch conditions, paying on average only a few microseconds when testing an input assignment. Hence, the time spent building the symbolic expressions can be amortized over thousands of (cheap) query evaluations, reducing the gap between the efficiency of a fuzzer and a concolic executor. Nonetheless, FUZZOLIC is still a hybrid fuzzer and it needs to run in parallel to a traditional fuzzer to provide good results, since some non-deterministic mutations, such as randomly combining inputs, are not performed by FUZZOLIC.

VI. THREATS TO VALIDITY AND LIMITATIONS

Floating-point arithmetic. Our current implementation of FUZZOLIC and FUZZY-SAT does not handle symbolic expressions involving floating-point operations. In case of floating-point instructions during program execution, FUZZOLIC concretizes the symbolic expressions. Although this is the same strategy adopted by QSYM, we acknowledge that it may harm

the effectiveness of the concolic executor on programs heavily based on floating-point computations.

Order of the mutations in FUZZY-SAT. The current implementation of FUZZY-SAT applies mutations using a specific order and stops as soon as one of the mutations is successful in finding a valid assignment for a query. In particular, FUZZY-SAT runs first the *cheapest* rules that are more *likely* able to succeed: e.g., given an input-to-state relation, trying the input-to-state rule first makes sense as it requires a few attempts and has high chances to succeed [32]. Hence, results reported in Section V are based on the order currently adopted by FUZZOLIC. An interesting experiment would be to evaluate how FUZZY-SAT would perform when changing the order of the mutations.

Impact of FUZZY-SAT in hybrid fuzzing. In Section V, we have investigated the impact of FUZZY-SAT inside two concolic executors: FUZZOLIC and QSYM. Our results are promising and suggest that FUZZY-SAT can be beneficial in the context of hybrid fuzzing. However, we believe it would be interesting to integrate FUZZY-SAT in other frameworks to have additional insights on its effect. For instance, the benefit from using FUZZY-SAT inside a concolic executor that is slow at building symbolic expression would be marginal as most of the analysis time would be spent in the emulation phase and not in the solving one. On the other hand, an efficient concolic executor should benefit from FUZZY-SAT as long as the number of queries submitted to the solver during an experiment is very high: if the number of queries is low, then a slower but more accurate traditional SMT solver would likely perform better than FUZZY-SAT.

VII. CONCLUSIONS

FUZZY-SAT is an approximate solver that uses fuzzing techniques to efficiently solve queries generated by concolic execution, helping hybrid fuzzers scale better on several real-world programs.

We have currently identified two interesting future directions. First, we plan to integrate FUZZY-SAT in the concolic execution framework SYMCC, which has been shown to be very efficient at building symbolic expressions and thus should benefit from using an efficient approximate solver. Second, we would like to devise an effective heuristic for dynamically switching during the exploration between FUZZY-SAT and a traditional SMT solver depending on the workload generated by the concolic executor on the solver backend.

REFERENCES

[1] M. Zalewski, "American Fuzzy Lop," https://github.com/Google/AFL, 2019, [Online; accessed 20-Aug-2020].

[2] M. Heuse, H. Eißfeldt, and A. Fioraldi, "AFL++," https://github.com/vanhauser-thc/AFLplusplus, 2019, [Online; accessed 20-Aug-2020].

[3] C. Barrett and C. Tinelli, *Satisfiability Modulo Theories*. Springer International Publishing, 2018, pp. 305–343.

[4] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18, 2018, pp. 745–761. [Online]. Available: http://dl.acm.org/citation.cfm?id=3277203.3277260

[5] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[6] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceeings of the 23nd Annual Network and Distributed System Security Symposium*, ser. NDSS'16, 2016. [Online]. Available: http://www.internetsociety.org/sites/default/files/blogs-media/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[7] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[8] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 521–532. [Online]. Available: https://doi.org/10.1145/3338906.3338921

[9] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-Scale Automated Vulnerability Addition," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, ser. SP '16, 2016, pp. 110–121. [Online]. Available: https://doi.org/10.1109/SP.2016.15

[10] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 736–747. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00082

[11] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*, 3rd ed. Hoboken and N.J: John Wiley & Sons, 2012.

[12] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, pp. 50:1–50:39, 2018. [Online]. Available: http://doi.acm.org/10.1145/3182657

[13] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The Fuzzing Book," https://www.fuzzingbook.org/, 2019, [Online; accessed 20-Aug-2020].

[14] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, ser. SP'16, 2016, pp. 138–157. [Online]. Available: http://dx.doi.org/10.1109/SP.2016.17

[15] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Trans. on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 2:1–2:49, 2012. [Online]. Available: http://doi.acm.org/10.1145/2110356.2110358

[16] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[17] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, 2008, p. 15–26. [Online]. Available: https://doi.org/10.1145/1390630.1390635

[18] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12, 2012, pp. 380–394. [Online]. Available: https://doi.org/10.1109/SP.2012.31

[19] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting Malware Analysis with Symbolic Execution: A Case Study," in *Proceedings of the 2017 Cyber Security Cryptography and Machine Learning*, ser. CSCML '17. Springer International Publishing, 2017, pp. 171–188. [Online]. Available: https://doi.org/10.1007/978-3-319-60080-2_12

[20] L. Borzacchiello, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Reconstructing C2 Servers for Remote Access Trojans with Symbolic Execution," in *Proceedings of the 2019 Cyber Security Cryptography and Machine Learning*, ser. CSCML '19. Springer International Publishing, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-20951-3_12

[21] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated Whitebox Fuzz Testing," in *Proceedings of the 2008 Network and Distributed System Security Symposium*, ser. NDSS'08, 2008. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf

[22] E. Coppa, D. C. D'Elia, and C. Demetrescu, "Rethinking Pointer Reasoning in Symbolic Execution," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17, 2017, pp. 613–618. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115671

[23] L. Borzacchiello, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Memory models in symbolic execution: key ideas and new thoughts," *Software Testing, Verification and Reliability*, vol. 29, no. 8, 2019. [Online]. Available: https://doi.org/10.1002/stvr.1722

[24] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1613–1627. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00063

[25] P. Yao, Q. Shi, H. Huang, and C. Zhang, "Fast bit-vector satisfiability," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '20, 2020, p. 38–50. [Online]. Available: https://doi.org/10.1145/3395363.3397378

[26] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2019.2941681

[27] "Google OSS-Fuzz: continuous fuzzing of open source software," https://github.com/google/oss-fuzz, 2019, [Online; accessed 20-Aug-2020].

[28] "Circumventing Fuzzing Roadblocks with Compiler Transformations," https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016, [Online; accessed 20-Aug-2020].

[29] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[30] B. Yadegari and S. Debray, "Bit-level taint analysis," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2014, pp. 255–264. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SCAM.2014.43

[31] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725. [Online]. Available: https://doi.org/10.1109/SP.2018.00046

[32] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[33] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, 2020. [Online]. Available: https://doi.org/10.1145/3395363.3397372

[34] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "SLF: Fuzzing without valid seed inputs," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 712–723. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00080

[35] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Signedness-agnostic program analysis: Precise integer bounds for low-level code," in *Proceedings of the 10th Asian Symposium on Programming Languages and Systems*, ser. APLAS '12, 2012, pp. 115–130. [Online]. Available: https://doi.org/10.1007/978-3-642-35182-2_9

[36] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," ser. PLDI '05. ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[37] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, "SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed)," in *Proceedings of the 14th ACM ASIA Conference on Computer and Communications Security*, ser. ASIACCS '19, 2019. [Online]. Available: https://doi.org/10.1145/3321705.3329819

[38] Google, "FuzzBench: issue #131," https://github.com/google/fuzzbench/issues/131, 2020, [Online; accessed 20-Aug-2020].

[39] M. Bynens, "Smallest possible syntactically valid files of different types," https://github.com/mathiasbynens/small, 2019, [Online; accessed 20-Aug-2020].

[40] D. Liew, "JFS: issue #4," https://github.com/mc-imperial/jfs/issues/4, 2019, [Online; accessed 20-Aug-2020].

[41] K. Serebryany, "libFuzzer: a library for coverage-guided fuzz testing," http://llvm.org/docs/LibFuzzer.html, 2015, [Online; accessed 20-Aug-2020].

[42] R. J. Stinnett, "JFS: issue #22," https://github.com/mc-imperial/jfs/issues/22, 2019, [Online; accessed 20-Aug-2020].

[43] "CmpLog instrumentation for QEMU inspired by Redqueen," https://aflplus.plus/features/, 2020, [Online; accessed 20-Aug-2020].

[44] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies*, ser. WOOT '20. USENIX Association, 2020.

[45] "Single-system parallelization," https://aflplus.plus/docs/parallel_fuzzing/, 2020, [Online; accessed 20-Aug-2020].

[46] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy*, ser. SP '18, 2018, pp. 679–696. [Online]. Available: http://dx.doi.org/10.1109/SP.2018.00040

[47] A. Fioraldi, "afl-qemu-cov," https://github.com/andreafioraldi/afl-qemu-cov, [Online; accessed 20-Aug-2020].