Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

TC 11 Briefing Papers

FUZZOLIC: Mixing fuzzing and concolic execution ☆

Luca Borzacchiello, Emilio Coppa, Camil Demetrescu*

Sapienza University of Rome, Italy

ARTICLE INFO

Article history:
Received 3 February 2021
Revised 3 May 2021
Accepted 6 June 2021
Available online 11 June 2021

Keywords:
Bug detection
Concolic execution
Fuzzing testing
SMT Solver
Hybrid fuzzing

ABSTRACT

In the last few years, a large variety of approaches and methodologies have been explored in the context of software testing, ranging from black-box techniques, such as fuzzing, to white-box techniques, such as concolic execution, with a full spectrum of instances in between. Using these techniques, developers and security researchers have been able to identify in the last decade a large number of critical vulnerabilities in thousands of software projects.

In this article, we investigate how to improve the performance and effectiveness of concolic execution, proposing two main enhancements to the original approach. On one side, we devise a novel concolic executor that can analyze complex binary programs while running under QEMU and efficiently produce symbolic queries, which could generate valuable program inputs when solved. On the other side, we investigate whether techniques borrowed from the fuzzing domain can be applied to solve the symbolic queries generated by concolic execution, providing a viable alternative to accurate but expensive SMT solving techniques. We show that the combination of our concolic engine, FUZZOLIC, and our approximate solver, FUZZY-SAT, can perform better in terms of code coverage than popular state-of-the-art fuzzers on a variety of complex programs and can identify different unknown bugs in several real-world applications.

1. 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 (Heuse et al., 2019; Zalewski, 2019). 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) (Barrett and Tinelli, 2018) 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 (Poeplau and Francillon, 2020; Yun et al., 2018). By systematically negating these conditions, it steers

☆ This paper is supported in part by EU's Horizon 2020 research and innovation programme (grant agreement No. 830892, project SPARTA).
* Corresponding author.
E-mail address: coppa@diag.uniroma1.it (E. Coppa).
https://doi.org/10.1016/j.cose.2021.102368
0167-4048/

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 (Stephens et al., 2016a; Yun et al., 2018). 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.

The symbolic exploration performed by a concolic executor can be logically split into two distinct phases: *emulation* and *reasoning*. In the former, the concolic executor builds symbolic expressions and queries while executing the program path for a given input. In the latter, the executor optimizes the queries and submits them to the SMT solver to possibly obtain satisfiable assignments for the symbolic inputs and thus generate alternative inputs for the program. In the context of hybrid fuzzing, the generated inputs are then made available to the coverage-guided fuzzers running in parallel in order to help them make further progress in their analysis.

Contributions. In this article, we attempt to speed up both phases performed by a concolic executor in order to improve the overall effectiveness of hybrid fuzzing when applied to real-world programs.

On the emulation side, we propose a novel concolic framework, called Fuzzolic, built on top of the binary translator QEMU, which can offer significant benefits in terms of performance and versatility with respect to the current state-of-the-art binary concolic executor Qsym.

On the reasoning side, we devise a new approximate solver, called Fuzzy-Sat, which can test the satisfiability of symbolic queries generated by concolic engines. Fuzzy-Sat borrows ideas from the fuzzing domain and avoids relying on accurate but costly SMT solvers. In particular, Fuzzy-Sat analyzes the expressions contained within a symbolic query and performs *informed* mutations to possibly generate new valuable inputs. Hence, our approximate solver can be used to replace classic SMT solvers in the context of hybrid fuzzing. To demonstrate the potential behind Fuzzy-Sat, we integrate it into two binary-based concolic frameworks, Fuzzolic and Qsym, as well as into the source-based concolic executor SymCC.

Our experimental results can be summarized as follows:

1. we evaluate the efficiency of Fuzzolic at building symbolic expressions and queries. We compare its performance to Qsym and SymQEMU, which similarly to Fuzzolic can analyze binary code, and to SymCC, a recently released concolic executor that has been shown to be extremely performant during emulation but requires the source code of an application.
2. we compare Fuzzy-Sat to the SMT solver Z3 (De Moura and Bjørner, 2008) and the approximate solver JFS (Liew et al., 2019) 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.
3. we evaluate Fuzzolic with Fuzzy-Sat on 12 real-world programs against state-of-the-art tools including the two fuzzers AFL++ (Heuse et al., 2019) and Eclipser (Choi et al., 2019), and the three hybrid fuzzers Qsym, SymQEMU, and SymCC, showing that Fuzzolic can reach higher code coverage than the competitors. Finally, we report some bugs that were found by Fuzzolic while testing it on different real-world applications available on mainstream Linux distributions.

To facilitate extensions of our approach, we will make our contributions available at:
https://season-lab.github.io/fuzzolic/

Conference version of this article. A preliminary version (Borzacchiello et al., 2021) of this article has appeared in the proceedings of the 43rd International Conference on Software Engineering (ICSE 2021). The paper version of this article has mainly presented the design of Fuzzy-Sat, omitting any discussion of Fuzzolic. In this manuscript, we extend the conference paper, presenting the novel aspects behind the design of Fuzzolic. Although Fuzzy-Sat and Fuzzolic can be used independently from each other, we provide a unified view where we present both projects, which have been designed and developed in parallel. In particular, this manuscript contains the following additional contributions and improvements:

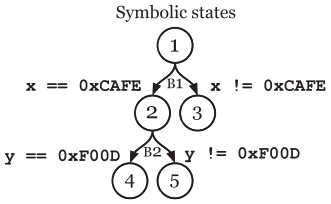
- the abstract and the introduction section have been revised to match the content of the journal version.
- Section 2 introduces the main ideas behind symbolic execution through the discussion of an example and discusses additional recent works in hybrid fuzzing.
- Section 3 is entirely new, presenting the design of Fuzzolic and providing a detailed high-level comparison with state-of-the-art concolic executors.
- Section 5.1 evaluates the efficiency of Fuzzolic at generating symbolic expressions on real-world benchmarks.
- Section 5.3 has been significantly extended to include additional experiments, such as comparisons with SymCC and SymQEMU, and to present a list of bugs that were identified by Fuzzolic while developing and testing it on real-world applications.
- Section 6 discusses several future directions that could provide valuable insights to the research community on how to extend Fuzzolic and Fuzzy-Sat.

2. Background

Fuzzolic and Fuzzy-Sat take inspiration from two popular software testing techniques (Myers et al., 2012): symbolic execution (Baldoni et al., 2018) and coverage-based grey-box fuzzing (Zeller et al., 2019). We now review the inner-workings of these two approaches, focusing on recent works that are tightly related to the ideas explored in this article.

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.,

```
  void foo() {
1.  int x = input();
2.  int y = input();
3.  if (x == 0xCAFE) {  // B1
4.    if (y == 0xF00D)  // B2
5.      crash();
6.    }
  }
```



- At branch B1 in state S1:
  - *true direction* → Q1: $x = 0\text{xCAFE}$
  - *false direction* → Q2: $x \neq 0\text{xCAFE}$
- At branch B2 in state in state S2:
  - *true direction* → Q3: $y = 0\text{xF00D} \wedge x = 0\text{xCAFE}$
  - *false direction* → Q4: $y \neq 0\text{xF00D} \wedge x = 0\text{xCAFE}$

Fig. 1 – Symbolic execution on a simple function `foo`.

[0, 255] for an unsigned byte). Symbolic execution 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, symbolic execution 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, symbolic execution 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.

An example of a symbolic exploration is provided in Fig. 1, where a simple C function is analyzed. Function `foo` takes two inputs, `x` and `y`, and performs equalities checks on their values. A symbolic engine starts the exploration from the beginning of the function and after evaluating the first two lines, it maps in the state S0 the two symbolic inputs α_x and α_y to the variables `x` and `y`, respectively. When evaluating the first branch instruction B1 at line 3, symbolic execution generates two SMT queries Q1 and Q2. Since there exist assignments that make both the queries satisfied, e.g., $\{\alpha_x = 0\text{xCAFE}, \alpha_y = 0\text{x0}\}$ for Q1 and $\{\alpha_x = 0\text{x0}, \alpha_y = 0\text{x0}\}$ for Q2, the engine forks the current execution, generating states S2 and S3. S3 has reached the end of the function and thus can be terminated. On the other hand, the executor evaluates branch B2 at line 4 for state S2 and generates the two queries Q3 and Q4. Since there exist assignments that make both the queries satisfied, e.g., $\{\alpha_x = 0\text{xCAFE}, \alpha_y = 0\text{xF00D}\}$ for Q3 and $\{\alpha_x = 0\text{xCAFE}, \alpha_y = 0\text{x0}\}$ for Q4, the engine forks again the current execution, generating states S4 and S5. S5 has reached the end of the function and thus can be terminated, while S4 can evaluate line 5, which triggers a crashing condition, such as a division by zero or an invalid memory operation. During the exploration, the engine can at any time dump the assignments from a branch query to produce concrete input values that can reproduce the program path followed by the state generating the query.

While our example considers a single function in C source code, symbolic execution can be used to analyze entire programs or even full operating systems, written in any representation of the program code, from high-level languages to low-level binary code. For instance, KLEE (Cadar et al., 2008) is one of the most popular source-based symbolic frameworks, that can analyze code in the LLVM intermediate representation, which can be obtained by compiling an application with the LLVM compiler toolchain. In the Java ecosystem, the analysis framework SPF (Pasareanu et al., 2008) is a standard solution for symbolically analyzing Java bytecode. At binary level, ANGR (Shoshitaishvili et al., 2016) is one of the most user-friendly and versatile symbolic frameworks that can analyze binary programs for many different platforms. While all of these frameworks are mainly designed for analyzing a single application, S^2E (Chipounov et al., 2012)) can scale symbolic execution to the entire program stack, including the operating system code.

Concolic execution. A twist of symbolic execution designed with scalability in mind is concolic execution (Godefroid et al., 2008), 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 of 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 (Yun et al., 2018) 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 but also generates inputs that can lead the program to access alternative memory locations. More recently, SYMCC (Poeplau and Francillon, 2020) has improved the design of QSYM by proposing a source-based instrumentation approach that further reduces the emulation time.

While revising this manuscript, a new concolic executor called SYMQEMU (Poeplau and Francillon, 2021) has been re-

leased by the same authors of SymCC: similarly to Qsym, this project can perform concolic execution on binary code without requiring to recompile the application with a custom compiler toolchain. Since this framework shares several design features with Fuzzolic, we discuss a detailed comparison in Section 3.6.

Approximate constraint solving. Many queries generated by concolic executors are either unsatisfiable or cannot be solved within a limited amount of time (Yun et al., 2018). 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 (Liew et al., 2019), 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 (Huang et al., 2020) and Trident (Yao et al., 2020), 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 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 symbolic execution 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) (Zalewski, 2019) 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 (Zalewski, 2019).

Hybrid fuzzing. Although CGF fuzzers have found thousands of bugs in the last years (Google, 2019; Pham et al., 2019), 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 (Stephens et al., 2016b) 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 benefit from the work done by each other.

Path prioritization and optimal search strategies. Optimizing how fuzzing and concolic execution cooperate in hybrid fuzzing has been investigated by several works during the last years (Liu et al., 2020; Wang et al., 2018; Zhao et al., 2019). MDPC (Wang et al., 2018) suggests to use *markov decision processes* to estimate the cost of constraint solving, helping hybrid fuzzing to optimally decide when to activate concolic execution. As the cost estimation can be expensive, DigFuzz (Zhao et al., 2019) proposes instead a probabilistic path prioritization strategy based on the Monte Carlo method. Of a different flavor is the proposal from LEGION (Liu et al., 2020), which regards the path exploration performed by hybrid fuzzing as a search space. LEGION uses concolic execution to identify alternative subtrees inside the path space and then exploits a custom form of directed fuzzing to generate inputs for a given subtree. These works are complementary to the contributions discussed in this article as these solutions still require performing concolic execution and solving path constraints. In the remainder of this article, we consider for Fuzzolic only the search strategy and setup originally proposed by Qsym (discussed in detail in Section 3.5), as it is adopted by several concolic executors and thus allow us to perform a fair comparison across different solutions.

Recent improvements in coverage-guided fuzzing. In the last few 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 (lafintel, 2016) splits multi-byte checks into single-byte comparisons, helping the fuzzer track the intermediate progress when reasoning on a branch condition. Vuzzer (Rawat et al., 2017) integrates dynamic taint analysis (DTA) (Yadegari and Debray, 2014) 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 (Chen and Chen, 2018) 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 (Aschermann et al., 2019) 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 in-

serts 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 (Fioraldi et al., 2020a) 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* (Pham et al., 2019) to effectively fuzz applications processing structured input formats.

SLF (You et al., 2019) 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 (Choi et al., 2019) 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.

3. Efficient generation of symbolic queries

Concolic execution has shown to be extremely effective when paired with coverage-guided fuzzing in what is called a hybrid fuzzing setup. However, the overhead of concolic execution when analyzing a single program path is still very high, limiting the scalability of the approach. In this section we present the inner-workings of a new framework called Fuzzolic, pinpointing the main design choices that we took in order to make it efficiently build symbolic expressions and queries. In Section 4, we discuss instead a new query solver that can be paired with Fuzzolic but also with other concolic frameworks and can help to reduce the time spent in the reasoning phase by concolic execution.

3.1. Architecture

Fig. 2 depicts the high-level architecture of Fuzzolic. To motivate and explain each component, we analyze the properties that our design is trying to reach:

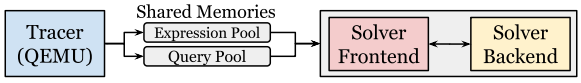


Fig. 2 – Internal architecture of Fuzzolic: a QEMU-based component executes the binary application, building symbolic expressions and queries within two shared memories (*Expression Pool* and *Query Pool*, respectively), while a solver frontend reads from the two shared memories, optimizing the expressions and submitting the queries to the solver backend.

- *Deployability*. Fuzzolic does not require the source code of an application to perform concolic execution over it. This makes it possible to run it with very little effort from the user on many applications, even proprietary ones. In particular, Fuzzolic builds on top of the dynamic binary translator QEMU (Bellard, 2005) that can efficiently analyze and execute binary code for many different platforms. Hence, our approach is inherently different from source-based symbolic frameworks, such as Klee and SymCC. While these approaches could potentially benefit from high-level code properties (e.g., knowledge on the size of a C `struct`) which are lost at binary level, they also come with practical disadvantages: they often require recompiling not only the source of the program but also any of its libraries using a specific compiler toolchain, which could require unexpected effort and force users to skip analysis over specific parts of an application. For instance, the developers of SymCC suggest (SymCC, 2020) to avoid symbolically analyzing the C library, relying on hand-written API models, which however could be highly inaccurate. Hence, binary- and source-based are complementary solutions that may fit different needs of the user.
- *Efficiency*. Fuzzolic exploits the Just-In-Time (JIT) compilation performed by QEMU to add instrumentation code that efficiently builds the symbolic expressions and branch queries at runtime. JIT compilation for concolic execution has been initially proposed by Qsym: its design has been experimentally shown (Poeplau and Francillon, 2019) to provide significant performance benefits compared to solutions that perform interpretation for generating the symbolic expressions for a given native instruction. For instance, the binary framework Angr, although extremely versatile and easy-to-extend, is well-known to incur slowdowns as high as 1000000× (Poeplau and Francillon, 2019). S2E is way more efficient than Angr but still performs interpretation over instructions handling symbolic data using Klee[2], thus still incurring a higher overhead compared to Qsym.
- *Decoupling*. The design of Fuzzolic overcomes one of the major problems affecting Qsym: Fuzzolic decouples the component based on QEMU that builds the symbolic expressions, dubbed *tracer* in Fuzzolic, from the component that reasons over these expressions, dubbed *solver*. This is

[2] To improve performance, the authors of S2E have expressed the will (S2E, 2018) to move away from Klee and directly perform symbolic interpretation of native code.

Table 1 – Comparison of FUZZOLIC with respect to six symbolic frameworks in terms of requirements for the analysis (works on binary code or requires the source code of an application), portability (low when tied to a platform, high when tied to a platform-independent IR), user effort (low when working on binary code, medium when requiring to recompile the code using a standard compiler toolchain, high when requiring a custom toolchain) and efficiency (low when performing interpretation, high when performing source or binary instrumentation, and medium when mixing interpretation and instrumentation). SYMQEMU is a project that has been presented and released while revising this manuscript.

FRAMEWORK	WORKS ON	PORTABILITY	USER EFFORT	EFFICIENCY
KLEE	Source	High	Medium	Low
SYMCC	Source	High	Medium	High
ANGR	Binary	High	Low	Low
S2E	Binary	Medium	Low	Medium
QSYM	Binary	Low	Low	High
SYMQEMU	Binary	High	Low	High
FUZZOLIC	Binary	High	Low	High

required as recent releases of most DBT frameworks, such as PIN (Luk et al., 2005) on which QSYM is based, 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. Unfortunately, SMT solvers are extremely complex pieces of software that could produce uncontrolled side effects. 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[3]. While QEMU does not yet impose this constraint, FUZZOLIC is already equipped with a design that should withstand future changes in this direction from the authors of this DBT framework. Hence, the tracing component and the solving component are executed within distinct processes in FUZZOLIC. In particular, the tracer runs under QEMU and generates symbolic expressions and queries using a compact representation, storing them into two shared memories that are also attached to the memory space of the solving component, which in turn translates the expressions into the Z3 language and submits the queries to the solver backend to generate alternative inputs.

- *Portability and extensibility*. While our current implementation is tuned to analyze user-mode code running on a Linux x86_64 platform, FUZZOLIC could be ported to other systems as QEMU is likely the most compatible and portable DBT, supporting a large variety of platforms. Indeed, our instrumentation targets the Tiny Code Generation (TCG) intermediate representation internally used by QEMU, which is platform-independent. Additionally, since QEMU can even perform system-mode analysis, emulating the entire operating system, FUZZOLIC could be extended in the future to scale its analysis even beyond the boundary of a single application. Notice that QSYM is based on PIN (Luk et al., 2005), a proprietary framework that supports mainly the x86/x86_64 architectures.

Table 1 summarizes a comparison of FUZZOLIC with respect to six popular symbolic frameworks: FUZZOLIC is unique as it can analyze binary code, making the effort for the user to run its analysis rather low; it is highly efficient as it performs just-in-time compilation of its instrumentation; it could be ported to other platforms thanks to the wide compatibility offered by QEMU.

[3] QSYM has been recently removed from the Google project FuzzBench due to its instability on recent Linux releases (Google, 2020).

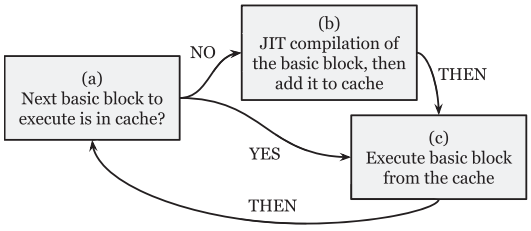


Fig. 3 – Execution workflow in QEMU: given the next basic block to execute (a), JIT compilation is performed if it is not in cache (b), then the block is executed (c).

During the revision of this manuscript, a new concolic executor based on QEMU has been released to the research community. As depicted in Table 1, this project – called SYMQEMU (Poeplau and Francillon, 2021) – shares several design aspects with FUZZOLIC. While both projects have the right to claim novelty on their design as they have been presented at the same time to the research community, in this manuscript we have chosen to first review in detail the most interesting aspects of FUZZOLIC that contribute at making it unique and then conclude providing a high-level comparison with SYMQEMU (Section 3.6) to help the reader understand the differences between these two projects. Additionally, Section 5 will present several experiments that compare the efficiency and effectiveness of the two frameworks.

3.2. *Just-in-time binary instrumentation*

A core component of FUZZOLIC is the tracer that is built on top of QEMU. Fig. 3 provides a high-level view of the execution workflow carried out by QEMU when executing a binary application. We can identify the following main steps:

(a) Given the address of the next basic block to execute[4], QEMU checks whether its native code is available in the JIT cache. When the block is not found, meaning that this

[4] Initially, QEMU starts from the entry point declared by the binary.

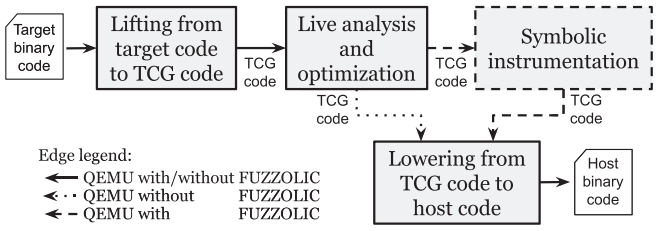


Fig. 4 – JIT compilation of a basic block in FUZZOLIC: given a basic block of the target platform, QEMU performs lifting to the architecture-independent intermediate representation TCG, the TCG code is then optimized, instrumented by FUZZOLIC, and finally lowered back to native code for the host platform. When the host and target architectures are different then QEMU is used as an emulator, when equal it is used just for binary instrumentation.

is the first time that QEMU is executing the code starting at the basic block address, it moves to step (b), otherwise to step (c).

(b) A basic block that was never executed is compiled, generating native code for the host platform where QEMU is running (which could be different with respect to the target architecture of the application binary when performing emulation), and inserted into the JIT cache. Finally, QEMU moves to step (c).

(c) QEMU executes the native code associated to the basic block, deriving the address of the next basic block to execute and then moving again to step (a). The only exception to this workflow is when the current basic block executes a system call, forcing QEMU to interact with the operating system before resuming the execution of the current block.

Step (b) is where FUZZOLIC intervenes to add instrumentation that generates symbolic expressions and queries at runtime. A crucial observation is that, while the same basic block may be executed by the program several times, the cost of compilation is only paid once for each basic block as QEMU maintains a JIT cache. Hence, when executing a basic block, FUZZOLIC does not need to parse the block instructions, e.g., disassemble them to understand their semantics, as done instead by other frameworks such as ANGR and KLEE. In order words, FUZZOLIC can move part of its analysis from basic block execution time to basic block compilation time.

Fig. 4 illustrates how QEMU compiles a target basic block into a host basic block. First the target code of the basic block is fetched from the binary, lifting each native instruction into an architecture-independent intermediate representation (IR) called Tiny Code Generation (TCG). As the lifting pass works in isolation over single instructions, QEMU devises a pass to optimize the entire TCG block using standard peephole techniques. After this pass, the standard QEMU moves to the lowering phase, where the TCG block is translated into native code that can run on the host platform. However, FUZZOLIC adds into the pipeline an additional pass for injecting symbolic instrumentation in the TCG block before lowering it to native code.

An alternative design choice regarding when to add the instrumentation would be to modify the lifting pass, generating symbolic instrumentation while translating a single target instruction into the TCG IR. Although this approach could initially appear to involve a lower implementation effort since QEMU currently does not make it straightforward to extend its compilation pipeline, it would come with two notable downsides. First, the changes into the QEMU code would be spread across many different functions of the lifter, making it harder to port the instrumentation to newer releases of QEMU as the lifter is one of the most sophisticated components that is constantly improved and extended. Second, the symbolic instrumentation cannot benefit from knowledge resulting from the analysis of the instructions of the entire block. For instance, it would be hard to determine whether a value read from memory using a symbolic address will be later used in the same block to override the instruction pointer, e.g., an indirect jump after reading from a switch table using a symbolic index[5].

The symbolic instrumentation added by FUZZOLIC comes into two flavors[6]:

- *Inline-only instrumentation*. Given one or more TCG instructions from the basic block, FUZZOLIC inserts TCG instructions into the block that manipulates at runtime the symbolic state without calling external functions. For instance, given the x86_64 instruction:

  ```
  movq    %rbx, %rax
  ```

  which moves 64 bits from register `%rbx` to register `%rax`, QEMU lifts it to the TCG IR:

  ```
  mov_i64 tcg_reg_2, tcg_reg_1
  ```

  that moves 64 bits from register `tcg_reg_1` to register `tcg_reg_2`; this IR code is instrumented by FUZZOLIC to:

  ```
  mov_i64 tcg_reg_symbolic_2, tcg_reg_symbolic_1
  mov_i64 tcg_reg_2, tcg_reg_1 # original code
  ```

  that moves the address of the symbolic expression[7] from the register `tcg_reg_symbolic_1` (which symbolically keeps track of `tcg_reg_1`) to the register `tcg_reg_symbolic_2` (which symbolically keeps track of `tcg_reg_2`).
- *Helper-based instrumentation*. Given one or more TCG instructions from the basic block, FUZZOLIC inserts TCG instructions into the block that manipulates at runtime the

[5] Given this information, a concolic executor may add instrumentation to generate a query able to produce alternative inputs that make the program cover additional entries within the switch table.

[6] We simplify some details in our examples for the sake of readability.

[7] If `tcg_reg_1` is constant then the expression is `NULL`.

Table 2 – Analysis modes in FUZZOLIC.

	ANALYSIS MODE		
	A	B	C
Expr. builder	✗	√	√
Branch queries	✗	✗	√
Overhead	low	medium	high
JIT cache	concrete	symbolic	symbolic
Use scenario	program startup and API models	uninteresting code	interesting code

symbolic state by calling helper functions defined by FUZZOLIC. For instance, the same x86_64 instruction discussed above could be instrumented also to:

```
call fuzzolic_move_reg,         # helper
        $0x0, $0,               # flags
        tcg_reg_symbolic_1,     # arg 1
        tcg_reg_symbolic_2_id   # arg 2
mov_i64 tcg_reg_2, tcg_reg_1 # original code
```

that would still move at runtime the address of the symbolic expression from `tcg_reg_symbolic_1` to `tcg_reg_symbolic_2` but requires QEMU to generate a call to the function `fuzzolic_move_reg`, i.e., a helper routine defined by FUZZOLIC.

Inline-only instrumentation is significantly more efficient than helper-based instrumentation as it does not require calling external code that forces QEMU to generate boilerplate code for preserving and restoring caller-save registers. However, inline instrumentation can be error-prone when handling complex operations and should not contain branching code to work correctly[8].

3.3. *Analysis modes*

When running a program under QEMU, FUZZOLIC dynamically switches between three analysis modes. These modes are tuned to minimize the overhead and/or improve the accuracy of the symbolic analysis. Table 2 summarizes the main differences between these three modes.

When the first mode is enabled (A), FUZZOLIC does not inject symbolic instrumentation into the running basic blocks, producing a runtime overhead that is comparable to the traditional QEMU. Hence, FUZZOLIC in this mode will not generate new symbolic expressions or submit any branch query to the solver. This mode is used by FUZZOLIC when executing basic blocks that either are known apriori to process only concrete data, or that are part of a well-known function, e.g., a routine from the C library, for which FUZZOLIC knows how to precisely model the side effects on the symbolic state without the need of instrumenting its code. The first use scenario happens for instance in the initial part of the program execution when no symbolic data has been yet generated by the program. The second use scenario is designed to help FUZZOLIC handle C functions whose semantics is well-known and for which an *API model* can be easily written: for instance, the function `strcmp` can be easily modeled by FUZZOLIC without symbolically analyzing the binary implementation, which could be highly optimized (e.g., based on vectorized instructions) and would bring FUZZOLIC to generate very complex and hard-to-solve queries. A variant of the previous use scenario is related to well-known functions whose side effects are not of interest: e.g., most programs do not care for the symbolic execution of output functions such as `printf` and instead can benefit when the analysis on their code is skipped[9]. A crucial difference with respect to source-based approaches, such as KLEE or SYMCC, that also exploit API models is that FUZZOLIC can always fallback to binary instrumentation whenever writing a model is too complex or not valuable, while source-based frameworks come with an *all-or-nothing* approach where you are forced to recompile an entire program module or to have models for all of its functions.

The second mode (B) is designed to handle basic blocks that should be symbolically instrumented to track their side effects on the symbolic state but that are of little interest in terms of code coverage for the end user. This mode is valuable when FUZZOLIC is used for generating inputs that increase the coverage on the application code and not for testing the implementation of standard libraries: in this scenario the user may configure FUZZOLIC to skip branch queries related to instructions contained in specific functions of the standard libraries. For instance, the user may want to skip branch queries generated by the heap allocator primitives, such as `realloc`, but still need to capture their side effects as they could move symbolic data when reallocating blocks. Notice that writing an accurate model for the heap allocator primitives can be extremely complex.

The third mode (C) is used when handling basic blocks of interesting code. In this mode, FUZZOLIC adds symbolic instrumentations and generates branch queries. The overhead of this analysis mode is the highest among all modes.

To govern the three analysis modes at runtime, FUZZOLIC switches between two JIT caches for basic blocks: one cache is used for basic blocks without any instrumentation and the other one for blocks containing symbolic instrumentation. Notice that the same basic block may be executed concretely and symbolically at different times during program execution, e.g., the same function could be executed at the beginning of the program when everything is concrete and then later on after injecting symbolic data in the program execution. Hence, the same basic block could be potentially available inside both JIT caches.

Switching between the three modes is done automatically by FUZZOLIC. The first mode is activated at the startup of the program and is maintained until the application invokes system calls that inject symbolic data into the computation. When some symbolic data reaches the computation, the third mode is activated and used for the remainder of the execution, only temporarily switching to the first or the second mode when *special* functions are executed by the program under analysis. In particular, the first mode is enabled by FUZZOLIC

[8] FUZZOLIC provides preliminary support for supporting even branching code within inline instrumentation.

[9] For instance, `printf` may perform checks on its arguments that may lead to concretizations in the symbolic state, possibly reducing the input space for the current state.

when the program executes a library function for which an API model is available. Similarly, FUZZOLIC enables the third mode when the program executes a library function for which an API model is not available but that has been marked[10] as uninteresting from the point of view of code coverage. In both cases, FUZZOLIC switches back to the third mode as soon as the *special* function returns to its caller.

3.4. *Interaction between tracer and solver*

As discussed in Section 3.1, FUZZOLIC is composed by two components, i.e., the tracer and the solver, that run into different processes and communicate through two shared memories (the expression pool and the query pool, respectively).

The tracer is based on QEMU and is in charge of analyzing the execution of a program, generating symbolic expressions and queries within the two shared memories. To minimize the memory usage and make the parsing process easy within the solving component, expressions and queries are represented by the tracer using two optimized custom C `structs`.

The solving component runs in parallel to the tracer and can be logically split into two subcomponents: the solver frontend and the solver backend. The solver frontend waits until a new query is pushed into the query pool and then translates it, as well as any expression involved in the query, into a Z3 query that can be submitted to the solver backend after a few optimization passes. The solver backend can be FUZZY-SAT, or the standard SMT solver Z3, and should reason over a query, possibly returning satisfiable assignments, allowing FUZZOLIC to generate alternate inputs.

When the analyzed program execution is terminated, the tracer pushes a final fake query in the pool to notify the solver that no new query will be generated afterwards and that it can terminate its execution as well as soon as it has finished processing the queries currently available in the pool.

3.5. *Hybrid fuzzing setup*

Hybrid fuzzing has proved that pairing symbolic execution and coverage-guided fuzzing can be beneficial. Hence, FUZZOLIC is optimized to run within the same hybrid setup that was originally proposed by QSYM.

Interaction with CGF. Fig. 5 depicts at high-level the setup used by FUZZOLIC when running in the hybrid fuzzing mode. This setup is characterized by one instance of FUZZOLIC, performing symbolic exploration, and two instances of a coverage-guided fuzzer configured with different running configurations[11]. By convention, the first instance of the fuzzer is dubbed *master*, while the other one is dubbed *slave*. FUZZOLIC is compatible with any coverage-guided fuzzer implementing a parallel mode, i.e., a synchronization procedure commonly offered by most fuzzers which makes a tool to maintain its input queue on the file system, hence accessible to other tools running in parallel, and to perform periodic synchronization with input queues of other running tools. The synchronization procedure is typically designed to make the tool import inside its own queue any input generated by other tools that is evaluated as interesting, e.g., any input that increases the code coverage when running the program. FUZZOLIC is designed to load inputs from an input queue of another tool (the slave fuzzer in our figure) and store inputs in one dedicated input queue. Hence other tools will eventually import inputs generated by FUZZOLIC and possibly mutate them. Notice that, since any interesting input generated by the master fuzzer will be eventually imported by the slave fuzzer, FUZZOLIC only loads inputs from the fuzzer slave to avoid analyzing duplicated inputs.

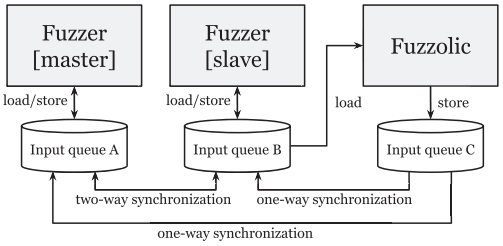


Fig. 5 – Hybrid fuzzing setup: FUZZOLIC runs in parallel with two instances of a coverage-guided fuzzer.

Filtering branch queries. While performing the exploration, a concolic executor may hit the same branch instruction several times, e.g., in presence of a loop condition. As the number of branches executed in a program could be very high, FUZZOLIC, similarly to previous hybrid fuzzers, avoids to perform a query over a branch condition whenever the branch instruction has been already met during a past exploration at the same execution context. In particular, when reaching a symbolic branch condition FUZZOLIC computes an hash value of the current instruction and the previous symbolic branch instructions met during the path exploration, taking into account also their calling context. The hash value is then used as an index to access a bitmap, which tracks whether the current branch condition has already led FUZZOLIC to submit a query to the solver in the past. When the entry of the bitmap is not set, FUZZOLIC submits the query to the solver and updates the bitmap. On the other hand, when the entry is set, the query is skipped. In other words, this strategy favors queries generated at different execution contexts.

3.6. *Discussion and comparison with SYMQEMU*

FUZZOLIC is a concolic executor optimized to run within a hybrid fuzzing setup that improves in different ways the original design proposed by QSYM.

The tracing component dynamically injects at runtime using QEMU the symbolic instrumentation in a program to build expressions and queries. To reduce the overhead, FUZZOLIC devises also three analysis modes that are dynamically enabled during the execution based on the running context.

[10] Currently, FUZZOLIC uses a manually-generated list of uninteresting functions. We plan in the future to devise heuristics that could build such a list.

[11] A common approach is to make the first instance perform deterministic mutations, while the second instance performs only non-deterministic mutations.

A unique design aspect of Fuzzolic is that it decouples the tracing component from the solving component, running them into two different processes and thus avoid execution interferences that may be hard to control for a dynamic binary translator.

While revising this manuscript, the concolic executor SymQEMU has been announced and released to the research community. Similarly to Fuzzolic, SymQEMU is based on QEMU and dynamically adds binary instrumentation during the program execution. However, the two projects take a different approach regarding *when* to add the binary instrumentation. As depicted by Fig. 4, Fuzzolic injects symbolic instrumentation after QEMU has generated the TCG code for an entire basic block. On the other hand, SymQEMU injects symbolic instrumentation when QEMU translates a single instruction into the TCG IR. As discussed in Section 3.2, the two approaches come with different implementation advantages and disadvantages: the approach taken by SymQEMU is easier to implement but possibly harder to port on newer releases (as it modifies the lifter in several points), while the approach taken by Fuzzolic is harder to implement but can reason on the instructions of an entire basic block, which may lead to better instrumentation.

Another important difference with respect to the instrumentation is that SymQEMU handles only architecture-independent TCG instructions, while Fuzzolic is able to symbolically instrument also a large number of architecture-dependent TCG *native helpers* for the x86 and x86_64 platforms. For instance, the native helpers are used by QEMU to model and correctly execute the instructions from the Streaming SIMD Extensions (SSE), which can be introduced by the compiler when optimizing the code of an application. More importantly, QEMU uses native helpers to model integer division on the x86 and x86_64 platforms. Hence, SymQEMU will not generate any symbolic expression in presence of TCG native helpers, omitting also to concretize the part of the symbolic state which may be affected by the handlers. Fuzzolic correctly handles the integer division helpers and the most common SSE/SSE2 helpers emitted by QEMU. Fuzzolic does not yet symbolically handle helpers related to atomic instructions and floating-point computations but concretizes the symbolic state to possibly avoid inconsistencies in the symbolic state.

Finally, SymQEMU does not devise the three analysis modes presented in Section 3.3 and does not decouple the solving component from the tracing component. While evaluating the benefits of the latter is hard as it is a design choice that could pay off only in the long run, in Section 5 we perform a comparison between the two frameworks to evaluate the impact of the former.

4. Efficient solving of symbolic queries

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 article, 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.

4.1. *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, Fuzzolic, and SymCC, are instead built on top of a few but essential primitives. In this article, 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 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.

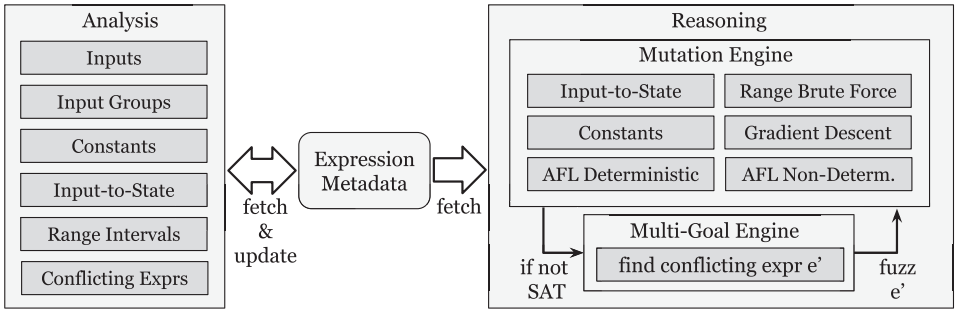


Fig. 6 – Internal architecture of FUZZY-SAT.

4.2. *Overview*

Architecture. To support the primitives presented in Section 4.1, the architecture of FUZZY-SAT (Fig. 6) has been structured around three main building blocks: the *analysis stage*, the *expression metadata*, and the *reasoning stage*.

The analysis stage (Section 4.3) 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 (Section 2) 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 (Section 4.4) 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.

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

Algorithm 1 – SOLVE implementation of FUZZY-SAT: analysis stage in light red, reasoning stage in cyan (initial mutations due to e in light cyan, multi-goal strategy in medium cyan, and optimistic solving in dark cyan).

```
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
```

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[12] and then *fixes*

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

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. In the remainder of this section, we focus on the internal details of functions Analyze (Section 4.3) and Mutate (Section 4.4), which are crucial core elements of Fuzzy-Sat.

4.3. 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.

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;

Examples: The expression $i_1 + i_0 == 0xABCD$ 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 (Section 2). 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.

Examples: 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 $i_1 \oplus 0xF0 == 0x0F$, Fuzzy-Sat collects the constants 0xF0, 0x0F, and 0xFF (i.e., $0xF0 \oplus 0x0F$) 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 com-

pactly and efficiently maintain these intervals, FUZZY-SAT uses *wrapped intervals* (Navas et al., 2012) 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 $(i_1 + i_0) + 0xAAAA <_{unsigned} 0xBBBB$, FUZZY-SAT computes the intervals $[0, 0x1110] \cup [0x5556, 0xFFFF]$ 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.

4.4. *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 (Section 4.2). 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.

Similarly to traditional SMT solvers, FUZZY-SAT can stop MUTATE after a user-defined time budget to avoid scalability issues in presence of hard-to-solve constraints. This time budget is set to 1 second in the current implementation.

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 (Section 4.3), 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 $i_1 + i_0 == 0xABCD$, FUZZY-SAT builds the assignment $\{i_0 \leftarrow 0xCD, i_1 \leftarrow 0xAB\}$. If the range interval over i_0 is $[0xDD, 0xFF]$ 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[13] 512.

Example: Given the expression $(i_1 + i_0) * 0xABCD == 0xCAFE$ 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.

Example: Given the expression $(i_1 + i_0) * 100 == 200$ 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 (Chen and Chen, 2018), e.g., $e' < e''$ can be transformed[14] into $f < 0$ with $f = e' - e''$.

The search algorithm implemented in FUZZY-SAT is inspired by ANGORA (Chen and Chen, 2018) 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

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

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

that satisfies the condition, e.g., given $i_0 < 1$, the assignment $\{i_0 \leftarrow 0x0\}$ satisfies the condition even if the global minimum for $i_0 - 1$ is given by $\{i_0 \leftarrow 0x81\}$. 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 $(i_0 + i_1) - 10 > (i_2 + i_3) - 5$ and a zero-filled input test case, then FUZZY-SAT transforms the expression into $((i_2 + i_3) - 5) - ((i_1 + i_0) - 10) < 0$, computes the gradients over the input groups $(i_1 + i_0)$ and $(i_2 + i_3)$, finding the assignment $\{i_0 \leftarrow 0x80, i_1 \leftarrow 0x06, i_2 \leftarrow 0x84, i_3 \leftarrow 0x01\}$ which makes the condition satisfied as $(0x80 + 0x06) - 10 = 32,764 > -31748 = (0x84 + 0x01) - 5$.

Deterministic and non-deterministic mutations. These two sets of input transformations are inspired by the two mutation stages from AFL (Section 2). 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 (Zalewski, 2019). 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).

4.5. *Implementation aspects*

FUZZY-SAT is written in C (10K LoC) and evaluates queries in the language devised by the Z3 Theorem Prover (De Moura and Bjørner, 2008). Similarly to traditional SMT solvers, it can be executed as a command-line tool providing as arguments an `smt2` file containing the query and an input seed from which FUZZY-SAT can extract concrete assignments, or it can be integrated into a concolic executor using C and C++ bindings that directly take as arguments a Z3 query and a buffer of raw bytes for the input seed.

Internally, to efficiently evaluate expressions using concrete assignments, FUZZY-SAT uses a fork of Z3 where the `Z3_model_eval` function has been optimized to efficiently deal with full concrete models.

4.6. *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.

5. Evaluation

In this section we address the following research questions:

- **RQ1**: How efficient is FUZZOLIC at building symbolic queries on real-world programs?
- **RQ2**: How effective and efficient is FUZZY-SAT at solving queries generated by concolic executors?
- **RQ3**: How do FUZZOLIC and FUZZY-SAT perform in a hybrid fuzzing setup?

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 (Google, 2019), and used in previous evaluations of state-of-the-art fuzzers (Aschermann et al., 2019; Choi et al., 2019; Fioraldi et al., 2020a; Poeplau and Francillon, 2020; Yun et al., 2018). As seeds, we use the AFL test cases (Zalewski, 2019), or when missing, minimal syntactically valid files (Bynens, 2019).

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, running Debian 9.2 x86_64.

5.1. *Efficiency of FUZZOLIC*

In this section we evaluate the efficiency of FUZZOLIC at generating symbolic queries, comparing its emulation time with respect to other state-of-the-art concolic executors. We remark that since different tools may be implemented quite differently, they may inherently generate very different queries that are hard to compare. Hence, to actually evaluate the effectiveness of a tool, we need to also take into account results from Section 5.3 that show whether a tool is able to generate valuable program inputs when reasoning on the queries generated during the emulation phase.

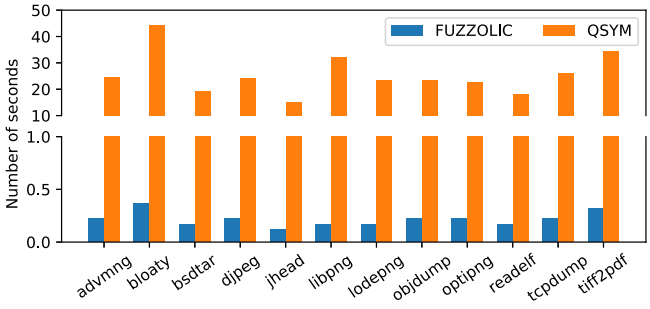


Fig. 7 – Time for building queries: FUZZOLIC vs QSYM.

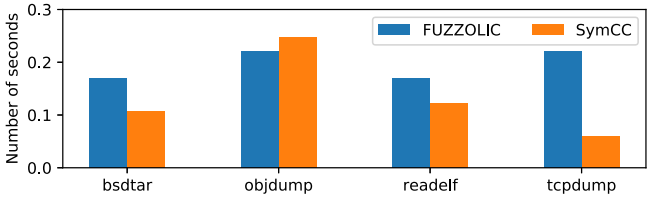


Fig. 8 – Time for building queries: FUZZOLIC vs SYMCC.

FUZZOLIC vs QSYM. Fig. 7 shows the average number of seconds taken by FUZZOLIC and QSYM over 100 runs to build symbolic queries for the 12 benchmarks on their initial seed. FUZZOLIC shows a significant speedup in the full set of benchmarks, with a time that is always smaller than 0.4 seconds compared to QSYM where the time is always larger than 14 seconds. This is not surprising as QSYM is still based on an outdated release of Intel PIN and its porting to newer releases has been proved to be challenging. Moreover, the instrumentation injected by QSYM is not particularly optimized, often incurring slowdowns that could be reduced by implementing PIN-specific tweaks.

FUZZOLIC vs SYMCC. A more challenging comparison is shown in Fig. 8, where the emulation time of FUZZOLIC is compared to the one of the source-based concolic executor SYMCC. Unfortunately, as SYMCC requires compiling the benchmarks with a specific compiler toolchain, we were able to correctly build and validate instrumented binaries only on four of our targets. Even on these four programs, we have to spend considerable time debugging and fixing issues[15] that result from the lack of API models in SYMCC or due to their inaccuracy (as they are hand-written).

FUZZOLIC is slower on 3 out of 4 benchmarks: this result is very promising as it shows that the gap between a source-based concolic executor and a binary concolic executor can be quite small, i.e., a slowdown less than 2× on average. On objdump, FUZZOLIC is faster than SYMCC thanks to the use of the optimized analysis modes described in Section 3.3. As these modes make FUZZOLIC alter the instrumentation based on the running context, SYMCC cannot easily devise them at compilation time.

Finally, we remark that since SYMCC is not instrumenting the full set of system libraries (e.g., the standard C library), FUZZOLIC is likely performing additional work compared to SYMCC. Unfortunately, comparing the number of expressions built by the two frameworks would not be fair as they may build the expressions in quite different ways, leading to very different numbers of expressions. Section 5.3 will help to understand whether the additional work done by FUZZOLIC can provide any benefit in terms of code coverage during hybrid fuzzing.

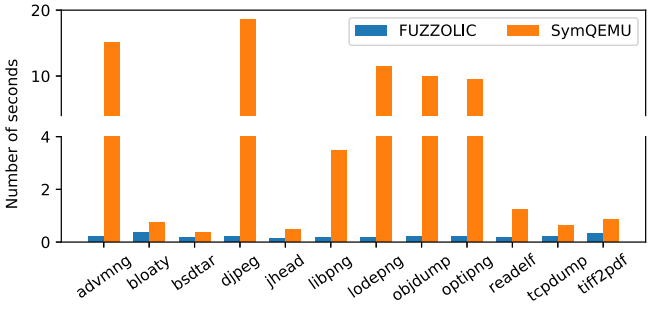


Fig. 9 – Time for building queries: FUZZOLIC vs SYMQEMU.

FUZZOLIC vs SYMQEMU. Another interesting comparison is between FUZZOLIC and SYMQEMU, as they share a similar design and are based on the same dynamic binary translator. Fig. 9 shows the emulation time of the two frameworks when running the 12 benchmarks on their initial seed. FUZZOLIC is always faster than SYMQEMU, which has an emulation time higher than 10 seconds on 5 out of 12 benchmarks. After performing performance profiling[16] on two benchmarks (djpeg and objdump), we believe that the bottleneck is due to how SYMQEMU is handling some TCG instructions that handle groups of bytes: FUZZOLIC contains a few low-level optimizations to efficiently handle *concat* and *extract* operations from groups of bytes. When considering the other benchmarks (7 out of 12), FUZZOLIC is likely faster than SYMQEMU thanks to two main factors: (a) inline instrumentation, which is done when handling some TCG operations, and (b) the optimized analysis modes.

Finally, we remark that since SYMQEMU is not instrumenting TCG native helpers (Section 3.6), e.g., helpers handling integer divisions on the x86_64 platform, FUZZOLIC is likely performing additional work compared to SYMQEMU. Similarly to SYMCC, we believe that comparing the number of expressions built by the two frameworks would not be fair as they generate them in quite different ways.

Impact of different analysis modes in FUZZOLIC. To assess the impact of the analysis modes in FUZZOLIC and to evaluate the cost of synchronization and communication with solver component via shared memories, we repeated the previous experiments using different configurations in FUZZOLIC. Fig. 10 shows the emulation time when:

- *Configuration X*. The tracer runs only using analysis mode C (the most expensive in terms of overhead, which is used by default in QSYM and SYMCC) and without any active shared memory, hence ignoring the cost of synchronization with the solver.

[15] GitHub issues and pull requests: (Borzacchiello, 2020k; 2021a; 2021b).

[16] We also tried to disable the garbage collector of SYMQEMU without observing any significant reduction in the emulation time.

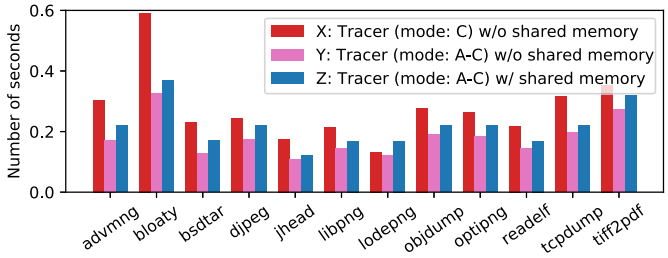


Fig. 10 – Time for building queries: FUZZOLIC with different configurations for the tracer component.

- *Configuration* Y. The tracer dynamically switches between modes A, B, C and runs without using any active shared memory.
- *Configuration* Z. The tracer dynamically switches between modes A, B, C, communicating the symbolic queries to the solver component via shared memories. This is the setup considered in Fig. 7 and Fig. 8.

When focusing only on the tracer, without enabling any communication with the solver, i.e., configurations X and Y, we can see that FUZZOLIC benefits from dynamically switching between the three analysis modes in all benchmarks, generating a mean speedup of 1.5×. To possibly reason over the symbolic queries, our design requires running in parallel the solver component: we compare the times from configurations Y and Z to assess the cost of synchronization between these two components. Overall, the mean measured slowdown is 1.2× which could be reasonable when taking into account the benefits in terms of maintainability that our design may offer.

5.2. *Solving effectiveness of FUZZY-SAT*

To evaluate how effective and efficient FUZZY-SAT is 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 (Dolan-Gavitt et al., 2016) 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 3 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 configured by 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 (Aschermann et al., 2019; Choi et al., 2019) 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 5.3.

Lastly, we can see in Table 3 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, Fig. 11a reports the number of queries solved by Z3 when using a timeout of 1 second and Fig. 11b 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 (Section 2), which however is based on a different design. When considering the queries collected on the 12 benchmarks, it can be seen in Fig. 11a 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 to the large num-

Table 3 – 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×

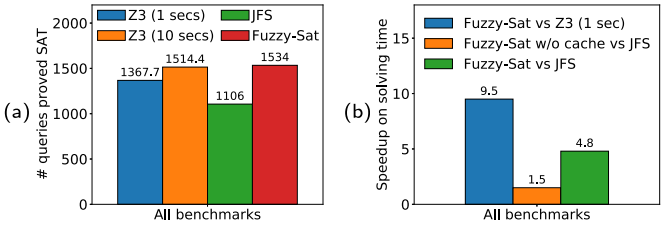


Fig. 11 – FUZZY-SAT vs other solvers on the 12 benchmarks: (a) number of queries proved satisfiable and (b) speedup on the solving time.

ber 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 (Liew, 2019), 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 (Serebryany, 2015), which does not integrate several fuzzing techniques that have inspired FUZZY-SAT.

When considering the solving time, FUZZY-SAT is 1.5× faster than JFS (Fig. 11b). However, when enabling analysis cache in FUZZY-SAT, the speedup increases up to 4.8×.

JFS does not currently provide a C interface (Stinnett, 2019), 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.

Impact of different mutations in FUZZY-SAT. An interesting question is which mutations contribute at making FUZZY-SAT effective. Table 4 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 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.

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 (Yun et al., 2018), looking for bugs within the four benchmarks base64, md5sum, uniq, and who. Table 5 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× 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 (Dolan-Gavitt et al., 2016), e.g., FUZZY-SAT has revealed 136 new bugs

Table 4 – 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%

Table 5 – 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)

on who. Since other works (Aschermann et al., 2019; Choi et al., 2019) reported a similar experimental observation, the additional bugs are likely not false positives.

5.3. *FUZZY-SAT and FUZZOLIC in hybrid fuzzing*

To further assess the effectiveness of FUZZY-SAT and FUZZOLIC, we evaluate these solutions in a hybrid setup, tracking the code coverage reached during 8-hour experiments (10 runs) considering the 12 benchmarks. In addition to FUZZOLIC, we consider three state-of-the-art binary fuzzing solutions: (a) AFL++ (Heuse et al., 2019) rev. `3f128` in QEMU mode, which integrates (AFLplusplus, 2020a) the colorization technique from REDQUEEN, as well as other improvements to AFL proposed by the fuzzing community during the last few years (Fioraldi et al., 2020b), (b) ECLIPSER rev. `b072f`, which devises one of the most effective *approximations* of concolic execution in literature, and (c) QSYM rev. `89a76`. Additionally, we also consider the source-based concolic executor SYMCC rev. `54e7f`, which is known to be quite efficient in the emulation phase during emulation, and the binary concolic executor SYMQEMU rev. `21c25c`, which shares several design choices with respect to FUZZOLIC.

To make a fair comparison across different tools, we consider them in the same hybrid setup: each tool runs in parallel with two instances (F_m, F_s) of a coverage-guided fuzzer, allowing periodic input queue synchronizations (AFLplusplus, 2020b). 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 yet not support a parallel mode, we extended AFL++ so that it correctly picks inputs from its queue.

Similarly to other works (Poeplau and Francillon, 2020; Yun et al., 2018), we plot in our charts the edge coverage and depict the 95% confidence interval using a shaded area.

FUZZOLIC with FUZZY-SAT vs other binary fuzzers. Fig. 12 shows the code coverage reached by the binary fuzzers on 8 out of 12 programs. On the remaining four benchmarks, the fuzzers reached soon a very similar coverage, making it hard to detect any significant trend and thus we omit their charts.

FUZZOLIC with FUZZY-SAT 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 edge coverage 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. In particular, the improvement results from the combination of an efficient solver (FUZZY-SAT) and an efficient tracer (FUZZOLIC). 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.

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 scale 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

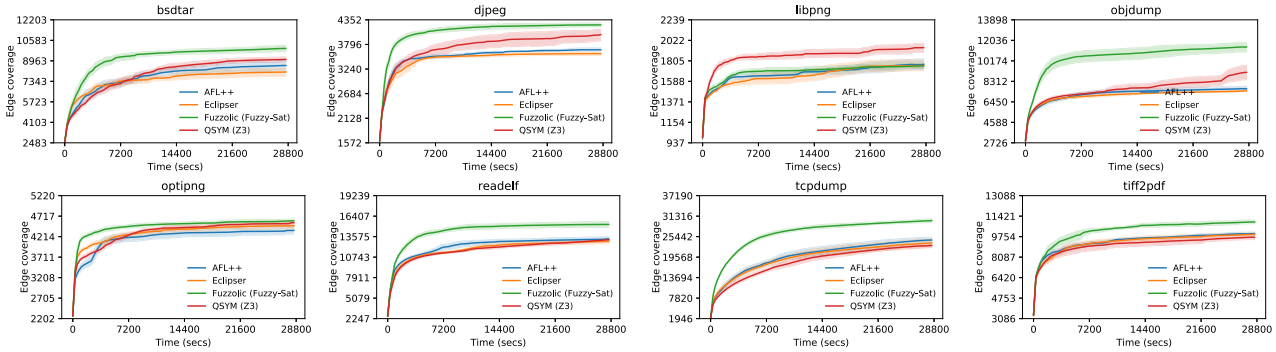


Fig. 12 – Edge coverage reached by FUZZOLIC with FUZZY-SAT vs four state-of-the-art binary fuzzers. The shaded areas are the 95% confidence intervals.

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, similarly to other concolic executors, FUZZOLIC is designed to run in parallel with a traditional fuzzer, since some non-deterministic mutations, such as randomly combining inputs, are not performed by FUZZOLIC but are likely to be extremely helpful when analyzing complex programs.

FUZZOLIC with FUZZY-SAT vs SYMCC. As observed in Section 5.1, SYMCC can be more efficient than FUZZOLIC during the emulation stage thanks to its source-based instrumentation. However, at the end of the day, concolic execution is relevant as long as it can build valuable queries: inconsistent queries are useless even when built very efficiently. For this reason, Fig. 13 shows a comparison on the four targets for which we were able to correctly build instrumented binaries for SYMCC. We can see that FUZZOLIC reaches higher code coverage in 3 out of 4 benchmarks, and performs similarly to SYMCC on bsdtar. This result is not completely unexpected as SYMCC does not ship with accurate models for several functions from the C library, potentially losing track of symbolic expressions during the program execution. Moreover, SYMCC will not symbolic analyze any other system library that has not been recompiled using its custom toolchain.

For instance, when comparing the number of queries generated by QSYM and SYMCC (these two frameworks use a similar runtime[17], making a comparison meaningful) we can observe that, on average when considering the four benchmarks, SYMCC is generating 34% less queries than QSYM. These numbers seem to confirm our claim that SYMCC is generating fewer queries than other concolic executors.

Hence, although SYMCC may be more efficient than FUZZOLIC at generating symbolic queries, it is currently less effective than FUZZOLIC due to the impracticality of recompiling an entire application, including its entire set of libraries. This claim has been confirmed by the authors of SYMCC: they have recently proposed SYMQEMU as a more practical solution.

FUZZOLIC: FUZZY-SAT vs Z3. An interesting question is related to the impact of FUZZY-SAT in FUZZOLIC and how this concolic executor would perform when using a standard SMT solver such as Z3. Fig. 14 shows the code coverage of FUZZOLIC on four benchmarks when using two different solver backends: FUZZY-SAT and Z3. FUZZY-SAT provides a significant benefit in terms of coverage in 3 out of 4 target programs. On readelf, while FUZZY-SAT initially gives an edge to FUZZOLIC compared to Z3, the SMT solver is able to close the gap before the end of the experiment. Overall, FUZZY-SAT appears to actually help FUZZOLIC improve its effectiveness.

QSYM: FUZZY-SAT vs Z3. Given the positive results observed on FUZZOLIC, we investigated the benefits from using FUZZY-SAT in QSYM. Fig. 15 shows how the edge coverage is affected when replacing Z3 with FUZZY-SAT in QSYM. Overall, we can see that FUZZY-SAT is generally able to increase the coverage of QSYM, however the amount of improvement changes is quite different depending on the specific benchmark under analysis. For instance, while on bsdtar and objdump we can observe that QSYM with FUZZY-SAT is now able to perform almost similarly to FUZZOLIC, it only slightly improves its coverage on readelf and tcpdump. This result is due to the limited scalability during the emulation phase of QSYM: e.g., on tcpdump the number of inputs analyzed by QSYM with FUZZY-SAT is 5.6× higher than when using Z3 but still 5.2× lower than FUZZOLIC.

SYMCC: FUZZY-SAT vs Z3. To further investigate the effectiveness of FUZZY-SAT, we have explored the impact of integrating this approximate solver in the source-based concolic executor SYMCC. Fig. 16 shows how the edge coverage changes in SYMCC when using FUZZY-SAT and Z3. Interestingly, only one benchmark (readelf) shows a significant improvement thanks to the use of FUZZY-SAT, while on the other targets SYMCC reaches very similar coverage regardless of the underlying solver.

Since an efficient concolic executor should theoretically benefit from a faster solver, we have inspected carefully these

[17] SYMCC has reused the runtime from QSYM, applying minimal changes.

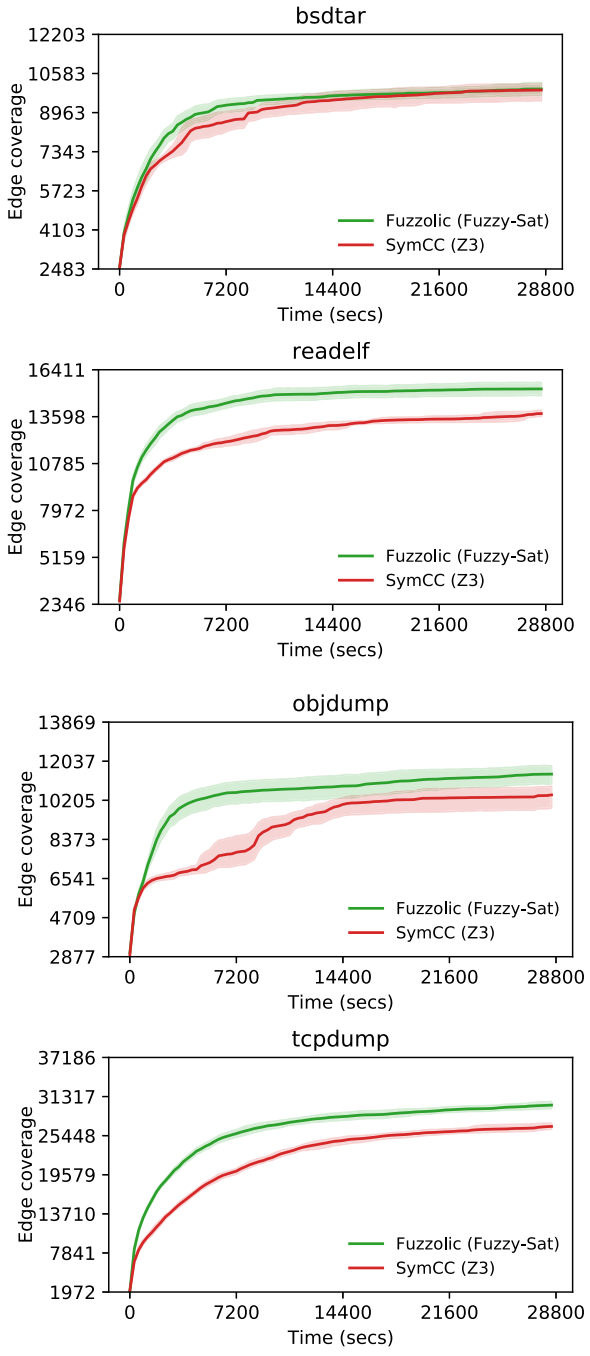


Fig. 13 – Edge coverage: FUZZOLIC vs SYMCC.

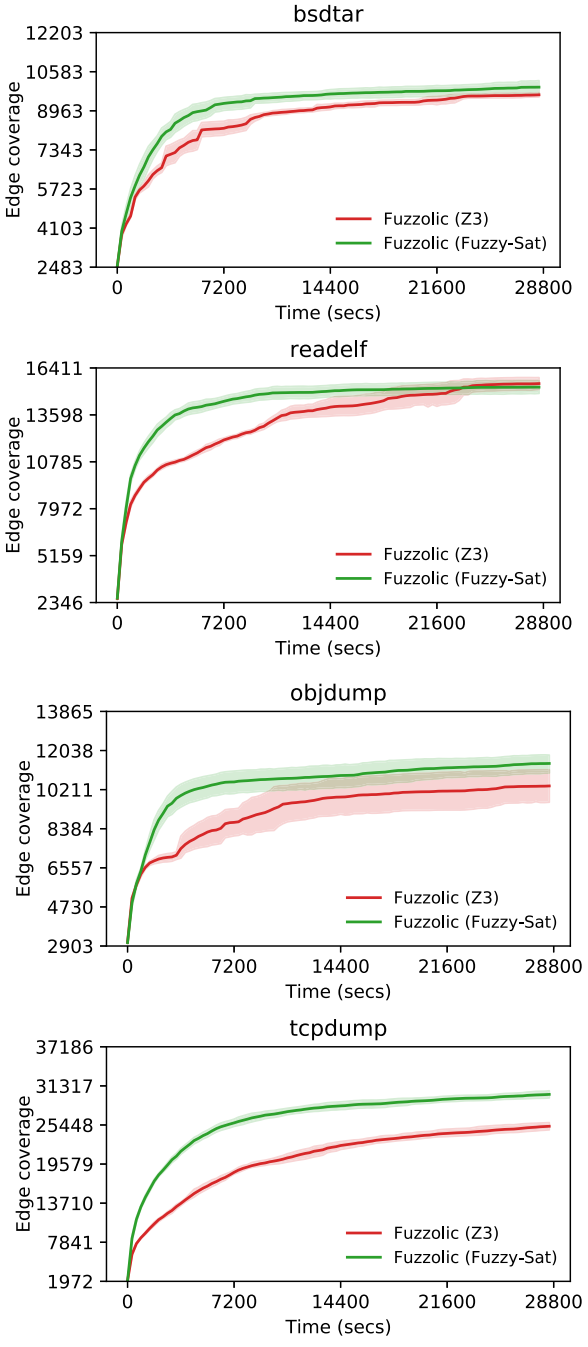


Fig. 14 – Edge coverage: impact of FUZZY-SAT in FUZZOLIC.

results, leading us to identify one main factor: as pointed out in the discussion covering the comparison between SYMCC and FUZZOLIC, SYMCC is generating a limited number of queries compared to other binary frameworks such as FUZZOLIC or QSYM. This is not unexpected as SYMCC is not symbolically instrumenting system libraries, thus possibly losing track of data flows that are generated in these libraries or that traverse code inside them.

One consequence of the reduced number of queries is that SYMCC is generating fewer inputs compared to, e.g., FUZZOLIC, reducing in turn the number of inputs pushed into the queue of the parallel fuzzer. As the parallel fuzzer gets fewer inputs inside its queue, it is less effective at generating new interesting inputs that could be picked later by the concolic executor. Overall, this effect is evident when considering that during our experiments SYMCC waits for long periods of time for new inputs from the fuzzer: hence, although FUZZY-SAT is able to reduce significantly the solving time of SYMCC, this concolic executor is not able to *reinvest* this *earned* time in other runs as it spends a large amount of time in waiting state, i.e., inactive. Indeed, the waiting state increases significantly when using FUZZY-SAT: e.g., +2780 seconds on bsdtar, +6205 seconds in objdump, and +9650 seconds in tcpdump. On readelf, the wait-

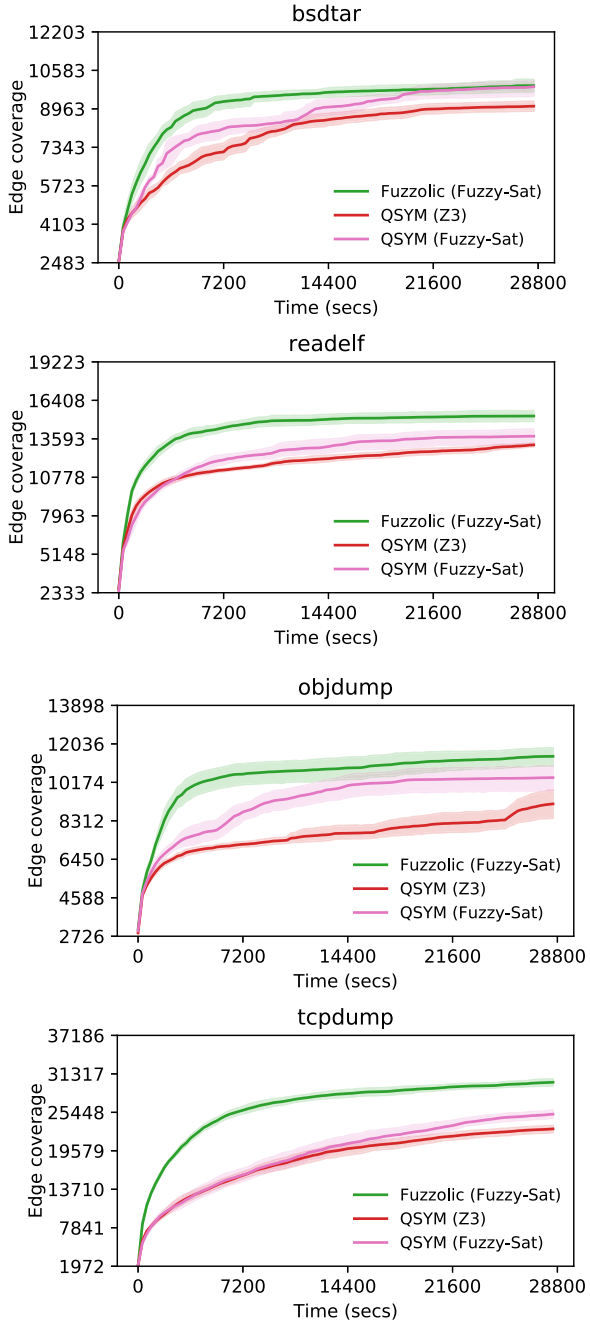


Fig. 15 – Edge coverage: impact of Fuzzy-Sat in Qsym.

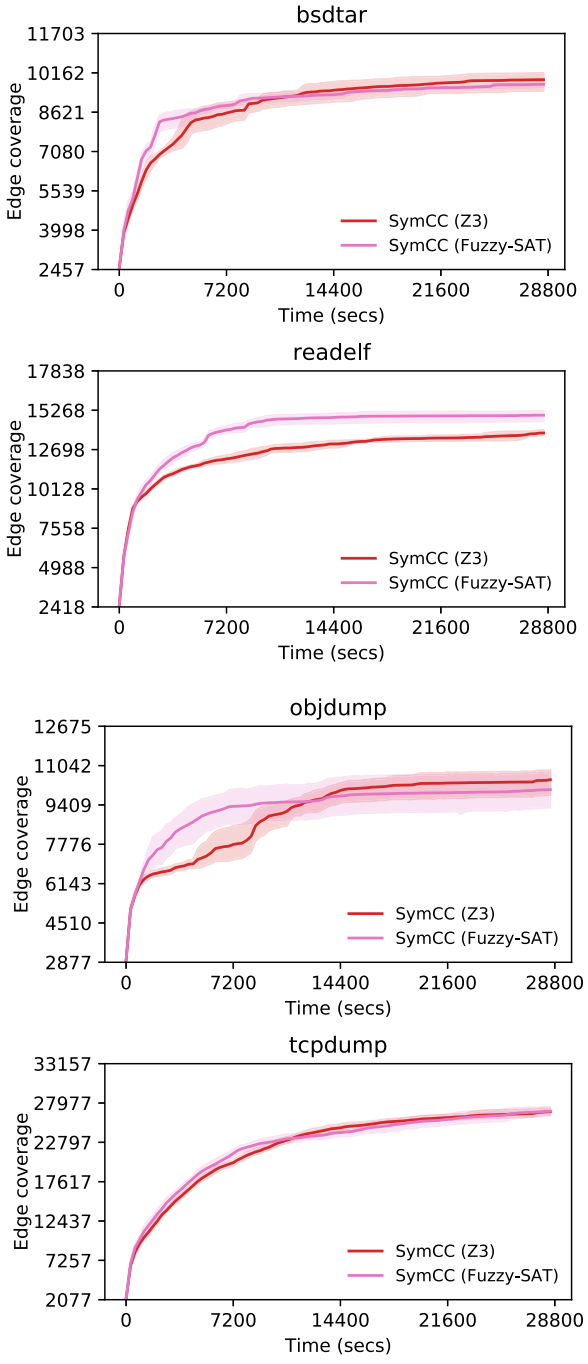


Fig. 16 – Edge coverage: impact of Fuzzy-Sat in SymCC.

ing state is very low (close to zero), showing that the concolic executor is active most of the time: in this scenario, Fuzzy-Sat shows a positive impact on the code coverage of SymCC, consistently with what has been observed in other concolic executors.

Overall, these experiments suggest that, when using Fuzzy-Sat, the hybrid fuzzing setup currently adopted by SymCC is suboptimal as it is under-utilizing the concolic executor: the waiting time could be spent running other complementary tools and analyses.

Fuzzolic vs SymQEMU. The last comparison that we consider is between Fuzzolic and SymQEMU. Fig. 17 shows the coverage over time on 8 out of the 12 benchmarks[18].

On three benchmarks (bsdtar, djpeg, and optipng), Fuzzolic and SymQEMU show a similar trend. On libpng, as already observed in previous comparisons, Fuzzolic with Fuzzy-Sat is worse than other concolic executors that use Z3, such as

[18] The other four benchmarks do not show any significant difference across the tools.

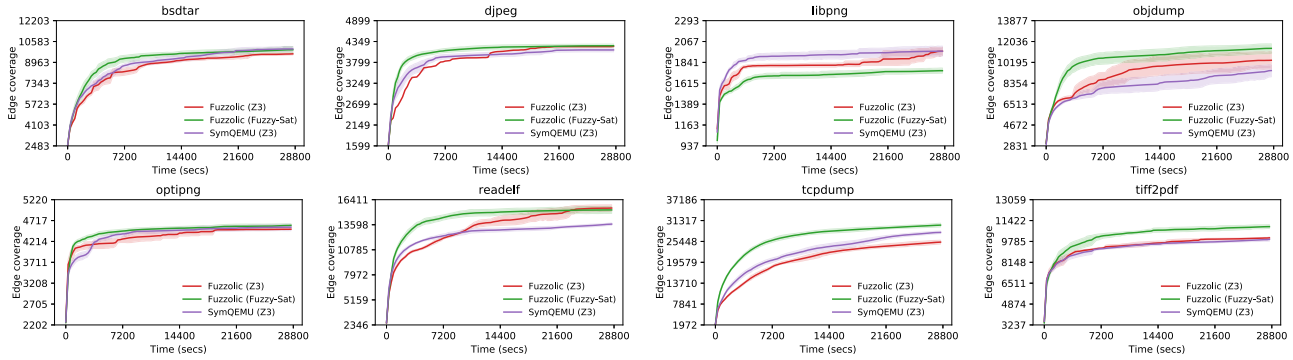


Fig. 17 – Edge coverage reached by FUZZOLIC (using FUZZY-SAT or Z3) vs SYMQEMU (using Z3).

SYMQEMU. On the other hand, FUZZOLIC with Z3 is able to close the gap with SYMQEMU before the end of the experiment.

On the remaining four benchmarks (objdump, readelf, tcpdump, and tiff2pdf) FUZZOLIC with FUZZY-SAT always achieves higher edge coverage than SYMQEMU. Interestingly, FUZZOLIC with Z3 has higher or comparable coverage than SYMQEMU on three of these benchmarks but shows an unexpected trend on tcpdump, where SYMQEMU outperforms FUZZOLIC with Z3. When investigating these results, we focused towards two main directions:

1. why does FUZZOLIC achieve higher coverage than SYMQEMU on several benchmarks?
2. why does FUZZOLIC with Z3 achieve lower coverage than SYMQEMU on tcpdump?

We have identified a few but critical limitations that currently affect SYMQEMU. As pointed out in Section 3.6, SYMQEMU is not instrumenting TCG native helpers, preventing it from reasoning correctly on several x86_64 instructions. Not only this makes it to ignore symbolic effects due to, e.g., SSE instructions (which could be inserted by a compiler when performing code optimizations) but also to ignore the effects of some traditional operations, such as integer divisions. For instance, SYMQEMU is not able to generate alternative inputs in case of this simplified code:

```
int foo(void) {
    int8_t x = get_input_byte();
    int8_t y = get_input_byte();
    if (x / y != 2) return 1;
    else return 0;
}
```

Another limitation is related to how SYMQEMU is reasoning in case of multiplications. For instance, its current instrumentation is not able to generate alternative inputs for this code:

```
int foo(void) {
    uint64_t a = get_input_byte();
    uint64_t b = get_input_byte();
    uint64_t c = (a * (unsigned __int128)b) >> 64;
    if (c != 2) return 1;
    else return 0;
}
```

While this code may seem exotic, it is actually common in real-world programs: for instance, most compilers for the x86_64 platform rewrite for performance reasons (Alverson, 1991) 64-bit divisions against constant values into 128-bit multiplications[19]. We remark that 128-bit multiplications are natively supported by the `mul` and `imul` instructions on the x86_64 platform. Unfortunately, SYMQEMU is currently unable to reason on the upper 64 bits of a 128-bit multiplication. Fixing this implementation problem is not trivial: SYMQEMU is adding some instrumentation that is later optimized away by QEMU (see Fig. 4).

Hence, when considering the first investigation direction, we believe that SYMQEMU is achieving a lower coverage due to these implementation limitations and due to its less efficient emulation phase (see Section 5.1).

When considering instead the second investigation direction, we carefully analyzed why FUZZOLIC with Z3 on tcpdump was under-performing compared to SYMQEMU considering that FUZZOLIC with FUZZY-SAT was instead outperforming SYMQEMU. The reason is due to the solving time: FUZZOLIC is building in most runs a few queries that cannot be solved by Z3 within the 10 seconds time budget. FUZZY-SAT is not able to solve most of these hard-to-solve queries but is able to *quickly fail*, without slowing down excessively the concolic executor. We then investigated why SYMQEMU is not experiencing the same slowdown even if it is using internally Z3: these hard-to-solve queries are not generated by SYMQEMU as they are related to divisions turned into 128-bit multiplications by the compiler. This benchmark shows one example where a concolic executor that is ignoring or wrongly handling part of the computation could be even faster than other more accurate concolic executors and still achieves decent code coverage. Nonetheless, the implementation issues in SYMQEMU have shown their negative impact on other benchmarks.

Real-world bugs identified by FUZZOLIC. We conclude our experimental evaluation by reporting our experience when running FUZZOLIC with FUZZY-SAT on the 12 benchmarks, as well as other command-line utilities available in the repository of mainstream Linux distributions. Although we did not perform a thorough and large-scale fuzzing campaign, we were still able to identify 29 bugs, out of which 10 were unknown to the program developers. The full list of bugs is reported in Table 6, where we also list the ones for which we did file new reports as we could find either existing bug re-

[19] One example could be `a / 5`, where `a` is a `uint64_t` variable.

Table 6 – Bugs found by FUZZOLIC with FUZZY-SAT on real-world programs. Some bugs can be reproduced only on the release used in our experiments since they have been fixed when considering the latest release (see FIXED UPSTREAM column). Some bugs, although reproducible even on the latest release, are not new/unknown (see NEW BUG column) since they have been already reported by third parties (see NOTE column) and thus we did not file new reports for them. For each new bug, we cite our reports in the NEW BUG column. Some new bugs have not yet been fixed by the developers/maintainers. For advmng, gocr, jhead, and optipng, we have considered the same releases used by ECLIPSER (Choi et al., 2019). Overall, we found 10 previously unknown bugs.

PROGRAM	FUZZED RELEASE	BUG DESCRIPTION	NEW BUG? [OUR REPORTS CITED]	FIXED UPSTREAM?	NOTE
advmng	2.1 Choi et al. (2019)	Invalid read in be_uint32_read	✗	√	Already fixed in the latest release
		Invalid read in png_print_chunk	✗	√	Already fixed in the latest release
		Invalid read in be_uint16_read	✗	√	Already fixed in the latest release
antiword	0.37-16 (DEBIAN)	Invalid read in vAnalyseDocumentSummaryInfo	√Borzacchiello (2020c)	✗	Maintainers asked for a patch
		Invalid read in bGet8DocumentText	√Borzacchiello (2020d)	✗	Maintainers asked for a patch
		Invalid read in vGet8Stylesheet	√Borzacchiello (2020e)	✗	Maintainers asked for a patch
grep	3.4	Invalid read in set_regs	✗	✗	Reported in 2016 Santiago Ruano Rincon (2020), still not fixed.
		Memory leak and performance regression w.r.t. grep-3.3	√Borzacchiello (2020i)	√	–
gocr	0.50 Choi et al. (2019)	Invalid read in context_correction	✗	√	Already fixed in the latest release
jhead	3.00--5 Choi et al. (2019)	Invalid read in process_EXIF	√Borzacchiello (2020f)	✗	No response yet from maintainers
		Use-after-free in show_IPTC	√Borzacchiello (2020g)	✗	No response yet from maintainers
		Invalid read in ProcessExifDir	✗	√	Already fixed in the latest release
		Invalid read in Get16u	✗	√	Already fixed in the latest release
mp42hls	1.6.0	Invalid read in AP4_StszAtom::GetSampleSize	✗	✗	Reported in 2017 Sarubbo (2017), still not fixed.
		Invalid read in AP4_MemoryByteStream::WritePartial	✗	✗	Reported in 2019 wcventure (2019), still not fixed.
		Invalid read in AP4_StszAtom::WriteFields	✗	✗	Identified in parallel by others seviezhou (2020).
		Invalid read in AP4_DescriptorListWriter::Action	✗	✗	Reported in 2020 natalie13m (2020), still not fixed.
		Invalid read in AP4_HvccAtom::AP4_HvccAtom	✗	✗	Reported in 2020 zhanggenex (2020), still not fixed.
mp4diff	1.6.0	Large memory allocation in AP4_RtpAtom::AP4_RtpAtom	✗	✗	Reported in 2019 zjuchenyuan (2019), still not fixed.
		Large memory allocation in AP4_Array::EnsureCapacity	✗	✗	Reported in 2019 wuk0n9 (2019), still not fixed.
objdump	2.34	Invalid free in _bfd_coff_free_symbols	✗	√	Identified in parallel by others Bugzilla (2020).
optipng	0.7.6 Choi et al. (2019)	Invalid write in LZWReadByte	✗	√	Already fixed in the latest release
scdoc	1.11.0--1 (DEBIAN)	Invalid read in parse_text	√ Borzacchiello (2020h)	√	–
sleuthkit	4.9.0	Invalid read in ntfs_dinode_lookup	√ Borzacchiello (2020j)	✗	No response yet from developers
		Invalid read in tsk_UTF16toUTF8	✗	✗	Reported in 2018 McGuire, Glenn (2018), still not fixed.
		strncpy parameter overlap in tsk_fs_name_copy	✗	✗	Reported in 2020 McGuire (2020), still not fixed.
		Invalid read in hfs_load_extended_attrs	✗	✗	Reported in 2017 adambuchbinder (2017), still not fixed.
xfpt	0.10-1 (DEBIAN)	Invalid read in read_process_macroline	√ Borzacchiello (2020a)	√	–
		Invalid write in dot_process	√ Borzacchiello (2020b)	√	–

ports or fixes that were not shipped in the release that we tested with FUZZOLIC. Overall, 24 bugs are related to invalid memory accesses (which could be confirmed by running the programs under Valgrind), one bug leads to an invalid `free` operation, one bug is a *use-after-free* vulnerability, one bug is a performance regression that also generates a memory leak, and finally the remaining two bugs make a program allocate an excessive amount of memory.

6. Conclusions

In this article, we have presented and experimentally analyzed FUZZOLIC and FUZZY-SAT.

FUZZOLIC is a novel concolic executor based on QEMU that can instrument binary programs at runtime in order to build symbolic expressions and queries. To reduce the runtime overhead and improve accuracy of the queries, it devises three analysis modes that are dynamically enabled during the program execution based on the running context. Moreover, differently from other concolic executors, FUZZOLIC runs the solver component, which reasons over the symbolic queries generated when analyzing a program, inside another process to reduce execution interferences that may be caused by the solver and negatively affect the analyzed application.

FUZZY-SAT is a new approximate solver for queries generated in the context of concolic execution. It analyzes each symbolic query and applies input mutations that are inspired by the software fuzzing realm. We successfully integrated FUZZY-SAT into three concolic execution frameworks: FUZZOLIC, QSYM, and SYMCC.

Our experiments have shown that FUZZOLIC is significantly faster than other binary concolic executors and can compete even with source-based approaches. FUZZY-SAT can be faster than traditional SMT solvers, such as Z3, or than other approximate solvers, such as JFS. The combination of FUZZOLIC and FUZZY-SAT can offer significantly benefits in terms of code coverage in a hybrid fuzzing setup compared to other state-of-the-art solutions. Finally, we have discussed our experience when running FUZZOLIC with FUZZY-SAT on different real-world programs, reporting that they are indeed able to identify several interesting unknown bugs.

As a main future research direction, we are exploring how to maintain a persistent state across different executions. This can be beneficial in the tracing component as well as in the solving component. We plan to elaborate more on this issue along two main lines.

First, we plan to implement a fork server, as seen in binary fuzzers such as AFL, that can maintain active the JIT cache across different runs and thus minimize the instrumentation time. In the current architecture of FUZZOLIC, the main challenge is to devise a strategy that can cope consistently with our three analysis modes.

Second, we plan to cache symbolic expressions and queries across different runs, reducing the time to optimize and translate them into Z3 expressions. Additionally, we plan to cache across different runs the analysis facts learned by FUZZY-SAT when analyzing the symbolic expressions. The main challenge for these changes is to devise an efficient mechanism to identify when the same expression is common to different execution paths.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Luca Borzacchiello: Conceptualization, Methodology, Software, Validation, Investigation, Writing - original draft. **Emilio Coppa:** Conceptualization, Methodology, Software, Validation, Investigation, Writing - original draft, Writing - review & editing. **Camil Demetrescu:** Conceptualization, Supervision, Project administration, Funding acquisition, Writing - original draft, Writing - review & editing.

REFERENCES

adambuchbinder, 2017. Sleuth Kit GitHub issue #912. https://github.com/sleuthkit/sleuthkit/issues/912. [Online; accessed 20-Aug-2020].

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

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

Alverson R. Integer division using reciprocals. Proceedings 10th IEEE Symposium on Computer Arithmetic, 1991. 186,187,188,189,190

Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T. REDQUEEN: fuzzing with input-to-state correspondence. Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS, 2019.

Baldoni R, Coppa E, D'Elia DC, Demetrescu C, Finocchi I. A survey of symbolic execution techniques. ACM Comput Surv 2018;51(3) 50:1–50:39. doi:10.1145/3182657.

Barrett C, Tinelli C. Satisfiability Modulo Theories. Springer International Publishing; 2018. p. 305–43.

Bellard F. Qemu, a fast and portable dynamic translator. Proceedings of the Annual Conference on USENIX Annual Technical Conference. Berkeley, CA, USA: USENIX Association, 2005. 41–41

Borzacchiello, L., 2020a. Debian Bug #968658. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=968658. [Online; accessed 22-Gen-2021].

Borzacchiello, L., 2020b. Debian Bug #968715. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=968715. [Online; accessed 22-Gen-2021].

Borzacchiello, L., 2020c. Debian Bug #968812. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=968812. [Online; accessed 22-Gen-2021].

Borzacchiello, L., 2020d. Debian Bug #968813. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=968813. [Online; accessed 22-Gen-2021].

Borzacchiello, L., 2020e. Debian Bug #968815. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=968815. [Online; accessed 22-Gen-2021].

Borzacchiello, L., 2020f. Debian Bug #968961. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=968961. [Online; accessed 22-Gen-2021].
Borzacchiello, L., 2020g. Debian Bug #968999. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=968999. [Online; accessed 22-Gen-2021].
Borzacchiello, L., 2020h. Debian Bug #969887. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=969887. [Online; accessed 14-Gen-2021].
Borzacchiello, L., 2020i. GNU Bug #43040. https://debbugs.gnu.org/cgi/bugreport.cgi?bug=43040. [Online; accessed 22-Gen-2021].
Borzacchiello, L., 2020j. Sleuth Kit GitHub issue #2012. https://github.com/sleuthkit/sleuthkit/issues/2012. [Online; accessed 22-Gen-2021].
Borzacchiello, L., 2020k. SymCC pull request #9. https://github.com/eurecom-s3/symcc/pull/9. [Online; accessed 25-Gen-2021].
Borzacchiello, L., 2021a. SymCC pull request #40. https://github.com/eurecom-s3/symcc/pull/40. [Online; accessed 25-Gen-2021].
Borzacchiello, L., 2021b. SymCC pull request #41. https://github.com/eurecom-s3/symcc/pull/41. [Online; accessed 25-Gen-2021].
Bugzilla, 2020. Bug #25447. https://sourceware.org/bugzilla/show_bug.cgi?id=25447. [Online; accessed 10-Sep-2019].
Bynens, M., 2019. Smallest possible syntactically valid files of different types. https://github.com/mathiasbynens/small. [Online; accessed 20-Aug-2020].
Borzacchiello L, Coppa E, Demetrescu C. Fuzzing Symbolic Expressions. Proceedings of the 43rd International Conference on Software Engineering. IEEE, 2021.
Cadar C, Dunbar D, Engler D. 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. Berkeley, CA, USA: USENIX Association; 2008. p. 209–24.
Chen P, Chen H. Angora: Efficient fuzzing by principled search. In: Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP); 2018. p. 711–25. doi:10.1109/SP.2018.00046.
Chipounov V, Kuznetsov V, Candea G. The S2E platform: design, implementation, and applications. ACM Trans. on Computer Systems (TOCS) 2012;30(1) 2:1–2:49. doi:10.1145/2110356.2110358.
Choi J, Jang J, Han C, Cha SK. Grey-box concolic testing on binary code. In: Proceedings of the 41st International Conference on Software Engineering; 2019. p. 736–47. doi:10.1109/ICSE.2019.00082.
De Moura L, Bjørner N. Z3: An efficient smt solver. In: Proceedings of 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems; 2008. p. 337–40. doi:10.1007/978-3-540-78800-3_24.
Dolan-Gavitt B, Hulin P, Kirda E, Leek T, Mambretti A, Robertson W, Ulrich F, Whelan R. LAVA: Large-Scale Automated Vulnerability Addition. In: Proceedings of the 2016 IEEE Symposium on Security and Privacy; 2016. p. 110–21. doi:10.1109/SP.2016.15.
Fioraldi A, D'Elia DC, Coppa E. WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), 2020.
Fioraldi A, Maier D, Eißfeldt H, Heuse M. AFL++: Combining incremental steps of fuzzing research. Proceedings of the 14th USENIX Workshop on Offensive Technologies, 2020.
Godefroid P, Levin MY, Molnar DA. Automated Whitebox Fuzz Testing. Proceedings of the 2008 Network and Distributed System Security Symposium, 2008.
Google, 2019. Google OSS-Fuzz: continuous fuzzing of open source software. https://github.com/google/oss-fuzz. [Online; accessed 20-Aug-2020].
Google, 2020. FuzzBench: issue #131. https://github.com/google/fuzzbench/issues/131. [Online; accessed 20-Aug-2020].
Heuse, M., Eißfeldt, H., Fioraldi, A., 2019. AFL++. https://github.com/vanhauser-thc/AFLplusplus. [Online; accessed 20-Aug-2020].
Huang H, Yao P, Wu R, Shi Q, Zhang C. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In: Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP); 2020. p. 1613–27. doi:10.1109/SP40000.2020.00063.
lafintel, 2016. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/. [Online; accessed 20-Aug-2020].
Liew, D., 2019. JFS: issue #4. https://github.com/mc-imperial/jfs/issues/4. [Online; accessed 20-Aug-2020].
Liew D, Cadar C, Donaldson AF, Stinnett JR. 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. New York, NY, USA: Association for Computing Machinery; 2019. p. 521–32. doi:10.1145/3338906.3338921.
Liu D, Ernst G, Murray T, Rubinstein BI. Legion: Best-first concolic testing. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering; 2020. p. 54–65.
Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. ACM; 2005.
McGuire, G., 2020. Sleuth Kit GitHub issue #1902. https://github.com/sleuthkit/sleuthkit/issues/1902. [Online; accessed 20-Aug-2020].
McGuire, Glenn, 2018. Sleuth Kit GitHub issue #1264. https://github.com/sleuthkit/sleuthkit/issues/1264. [Online; accessed 20-Aug-2020].
Myers GJ, Sandler C, Badgett T. The art of software testing. 3rd ed. Hoboken and NJ: John Wiley & Sons; 2012.
natalie13m, 2020. Bento4 GitHub issue #509. https://github.com/axiomatic-systems/Bento4/issues/509. [Online; accessed 20-Aug-2020].
Navas JA, Schachte P, Søndergaard H, Stuckey PJ. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In: Proceedings of the 10th Asian Symposium on Programming Languages and Systems; 2012. p. 115–30. doi:10.1007/2305978-3-642-35182-2_9.
Pasareanu CS, Mehlitz PC, Bushnell DH, Gundy-Burlet K, Lowry M, Person S, Pape M. 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; 2008. p. 15–26. doi:10.1145/1390630.1390635.
Pham V, Boehme M, Santosa AE, Caciulescu AR, Roychoudhury A. Smart greybox fuzzing. IEEE Trans. Software Eng. 2019. doi:10.1109/TSE.2019.2941681.
Poeplau S, Francillon A. SymQEMU: Compilation-based symbolic execution for binaries. Proceedings of the 2021 Network and Distributed System Security Symposium, 2021.
Poeplau S, Francillon A. Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and its Generation. Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC) 2019, 2019. San Juan, Puerto Rico

Poeplau S, Francillon A. Symbolic execution with symcc: Don't interpret, compile!. In: Proceedings of the 29th USENIX Security Symposium (USENIX Security 20). USENIX Association; 2020. p. 181–98.
Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. In: 24th Annual Network and Distributed System Security Symposium, NDSS. Vuzzer: Application-aware evolutionary fuzzing; 2017.
S2E, 2018. S2E. https://github.com/S2E/s2e-env/issues/178. [Online; accessed 20-Aug-2020].
Santiago Ruano Rincon, 2020. GNU Bug #22793. https://debbugs.gnu.org/cgi/bugreport.cgi?bug=22793. [Online; accessed 22-Gen-2021].
Sarubbo, A., 2017. Bento4 GitHub issue #204. https://github.com/axiomatic-systems/Bento4/issues/204. [Online; accessed 20-Aug-2020].
Serebryany, K., 2015. libFuzzer: a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html. [Online; accessed 20-Aug-2020].
seviezhou, 2020. Bento4 GitHub issue #546. https://github.com/axiomatic-systems/Bento4/issues/546. [Online; accessed 20-Aug-2020].
Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng S, Hauser C, Kruegel C, Vigna G. SOK: (state of) the art of war: Offensive techniques in binary analysis. In: Proceedings of the 2016 IEEE Symposium on Security and Privacy; 2016. p. 138–57. doi:10.1109/SP.2016.17.
Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G. In: Proceeings of the 23th Annual Network and Distributed System Security Symposium, NDSS. Driller: Augmenting fuzzing through selective symbolic execution.; 2016.
Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G. In: Proceeings of the 23nd Annual Network and Distributed System Security Symposium. Driller: Augmenting fuzzing through selective symbolic execution; 2016.
Stinnett, R. J., 2019. JFS: issue #22. https://github.com/mc-imperial/jfs/issues/22. [Online; accessed 20-Aug-2020].
SymCC, 2020. SymCC. https://github.com/eurecom-s3/symcc/blob/master/docs/Libc.txt. [Online; accessed 20-Aug-2020].
Wang X, Sun J, Chen Z, Zhang P, Wang J, Lin Y. Towards optimal concolic testing. In: Proceedings of the 40th International Conference on Software Engineering; 2018. p. 291–302. doi:10.1145/3180155.3180177.
wcventure, 2019. Bento4 GitHub issue #353. https://github.com/axiomatic-systems/Bento4/issues/353. [Online; accessed 20-Aug-2020].
wuk0n9, 2019. Bento4 GitHub issue #361. https://github.com/axiomatic-systems/Bento4/issues/361. [Online; accessed 20-Aug-2020].
Yadegari B, Debray S. Bit-level taint analysis. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM); 2014. p. 255–64. doi:10.1109/SCAM.2014.43.
Yao P, Shi Q, Huang H, Zhang C. Fast bit-vector satisfiability. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis; 2020. p. 38–50. doi:10.1145/3395363.3397378.
You W, Liu X, Ma S, Perry D, Zhang X, Liang B. SLF: Fuzzing without valid seed inputs. In: Proceedings of the 41st International Conference on Software Engineering; 2019. p. 712–23. doi:10.1109/ICSE.2019.00080.
Yun I, Lee S, Xu M, Jang Y, Kim T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In: Proceedings of the 27th USENIX Conference on Security Symposium; 2018. p. 745–61.
Zalewski, M., 2019. American Fuzzy Lop. https://github.com/Google/AFL. [Online; accessed 20-Aug-2020].
Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C., 2019. The Fuzzing Book. https://www.fuzzingbook.org/. [Online; accessed 20-Aug-2020].
zhanggenex, 2020. Bento4 GitHub issue #534. https://github.com/axiomatic-systems/Bento4/issues/534. [Online; accessed 20-Aug-2020].
Zhao L, Duan Y, Yin H, Xuan J. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. Proceedings of the 26th Annual Network and Distributed System Security Symposium, 2019.
zjuchenyuan. In: [Online; accessed 20-Aug-2020]. Bento4 github issue #386; 2019. https://github.com/axiomatic-systems/Bento4/issues/386

Luca Borzacchiello. Luca Borzacchiello is a Ph.D. student in Engineering in Computer Science at Sapienza University of Rome. He has been a partecipant of CyberChallenge.IT 2017. His research interests include cybersecurity, vulnerability detection, and program analysis.

Emilio Coppa. Emilio Coppa obtained his Ph.D. in Computer Science in 2015 from Sapienza University of Rome. He is currently an assistant professor at Sapienza. His research interests include software testing, vulnerability analysis, and reverse engineering techniques.

Camil Demetrescu. Camil Demetrescu is a full professor at Sapienza University of Rome. His research interests span different fields including computer security, algorithmics, and software analysis. He has been the national coordinator of the CyberChallenge.IT initiative supported by the Italian government.