The Journal of Systems and Software 211 (2024) 112001

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Testing concolic execution through consistency checks☆

Emilio Coppa [a,*], Alessio Izzillo [b]

[a] Luiss Guido Carli University, Italy
[b] Sapienza University of Rome, Italy

ARTICLE INFO

Keywords:
Symbolic execution
Testing
Concolic execution

ABSTRACT

Symbolic execution is a well-known software testing technique that evaluates how a program runs when considering a *symbolic* input, i.e., an input that can initially assume any concrete value admissible for its data type. The dynamic twist of this technique is dubbed *concolic execution* and has been demonstrated to be a practical technique for testing even complex real-world programs. Unfortunately, developing concolic engines is hard. Indeed, an engine has to correctly instrument the program to build accurate symbolic expressions, which represent the program computation. Furthermore, to reason over such expressions, it has to interact with an SMT solver. Hence, several implementation bugs may emerge within the different layers of an engine.

In this article, we consider the problem of testing concolic engines. In particular, we propose several testing strategies whose main intuition is to exploit the concrete state kept by the executor to identify inconsistencies within the symbolic state. We integrated our strategies into three state-of-the-art concolic executors (SymCC, SymQEMU, and Fuzzolic, respectively) and then performed several experiments to show that our ideas can find bugs in these frameworks. Overall, our approach was able to discover more than 12 bugs across these engines.

1. Introduction

Finding bugs in real-world applications is a crucial step during software development and maintenance. To this aim, several software testing techniques have been proposed during the latest decades (Myers et al., 2011) and a large number of studies have proved that they are indeed an essential means to improve the correctness, reliability, and security, of real-world software.

Symbolic execution. One advanced software testing technique is *symbolic execution* (Cadar and Sen, 2013a; Baldoni et al., 2018) which relies on the idea of not fixing a priori the input values for a program but instead evaluating how the program may behave when the program inputs change. It thus replaces each program input with a *symbolic* value that can initially assume any concrete value admissible for its data type. The evaluation of the program behavior is performed line by line, e.g., using an interpreter of the program code. When a computation involving the inputs is met, a new symbolic expression is built to represent the result of the computation as a function of the symbolic inputs. When the program execution reaches a branch statement, a symbolic executor forks the state, continuing the evaluation along each alternative path. To keep the execution state consistent with the branch conditions assumed to be true (or false) along a path, the engine tracks the *path constraints*. At any time during the exploration of the paths, the symbolic executor can use an SMT solver (De Moura and Bjørner, 2011) to: (a) validate whether a path is feasible, i.e., there exists an assignment for the program inputs which satisfies the path constraints, and (b) obtain one possible assignment for the program inputs, allowing a user to reproduce one of the considered paths when running concretely the program.

Symbolic execution has been shown to be extremely valuable for a large number of tasks, including vulnerability detection, malware reverse engineering, and software exploitation (Angelini et al., 2019; Borzacchiello et al., 2022, 2019). Examples of symbolic execution frameworks are KLEE (Cadar et al., 2008a), angr (Shoshitaishvili et al., 2016), manticore (Mossberg et al., 2019), and Symbolic Path Finder (Pasareanu et al., 2008).

Concolic execution. *Concolic execution* is the dynamic flavor of symbolic execution. This approach is based on the idea of mixing symbolic and concrete execution, which has been originally proposed in works such as DART (Godefroid et al., 2005), CUTE (Sen et al., 2005), and SAGE (Godefroid et al., 2008). However, in the last two decades, the research community has investigated several different flavors of concolic execution (Cadar and Sen, 2013b). In this article, we focus on the flavor adopted by several recent concolic frameworks, including QSYM (Yun et al., 2018), SymCC (Poeplau and Francillon, 2020), Fuzzolic (Borzacchiello et al., 2021b), SymQEMU (Poeplau and Francillon, 2021), SymFusion (Coppa et al., 2022), and SymSan (Chen

☆ Editor: Dr. Burak Turhan.

* Corresponding author.
E-mail addresses: ecoppa@luiss.it (E. Coppa), izzillo@diag.uniroma1.it (A. Izzillo).

https://doi.org/10.1016/j.jss.2024.112001
Received 4 May 2023; Received in revised form 1 February 2024; Accepted 10 February 2024
Available online 15 February 2024
0164-1212/© 2024 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

et al., 2022). First, these frameworks pick concrete values for the program inputs and run the program using them as in a *traditional* native execution. While the program is running, however, the concolic engine performs the symbolic evaluation along the taken program path. At each branch, the concolic executor exploits an SMT solver to possibly generate concrete input values able to make the program take the alternative branch direction with respect to the current one followed by the analyzed path. The generated inputs can then be used to start other concolic executions.

An important trait of concolic execution is that the program is *concretely* executed while analyzing the path, possibly bringing several benefits. First, concrete values can be implicitly maintained by the native execution, reducing substantially the work for the concolic executor. Second, whenever the concolic executor does not want to symbolically analyze a complex piece of code (e.g., a function from a system library) or cannot analyze it (e.g., code in kernel space), then the concolic executor can rely on the concrete state of the execution, trading accuracy in exchange of scalability and practicality. Finally, concolic executors can easily track the program behavior through code instrumentation, which can nowadays be added into a program using several mature frameworks: for instance, SymCC uses an LLVM pass (Lattner and Adve, 2004) to add instrumentation at compile time, while Fuzzolic (Borzacchiello et al., 2021b) exploits the JIT engine from QEMU (Bellard, 2005) to add instrumentation at running time.

Implementation complexity of symbolic frameworks. Implementing symbolic and concolic executors is quite complex. Indeed, operations performed by the program should be correctly represented with symbolic expressions. These symbolic expressions may be optimized by the executor to keep them simpler and more compact. Moreover, to reason over symbolic expressions, an executor relies on an SMT solver, requiring to translate the symbolic expressions into a solver-specific language. Finally, when the symbolic evaluation is omitted for a piece of code, the executor has to keep the symbolic state consistent by first inferring and then reproducing the side effects of the skipped code or, at least, perform *concretizations*, i.e., fixing the value of some input bytes, to avoid to reason on infeasible execution states.

Unfortunately, identifying implementation errors within a symbolic or concolic executor is extremely hard. Indeed, even a single symbolic path is used by the executor to represent a possibly large set of concrete executions, i.e., all the executions that would follow the same path. This characteristic makes it hard for a developer to reason on the correctness of the symbolic state, the symbolic expressions, and the results obtained from the solver, since the developer has to reason on the behaviors of many executions.

Existing approaches for testing symbolic frameworks. The research community, to the best of our knowledge, has not thoroughly investigated methodologies able to specifically support a developer when testing symbolic or concolic execution frameworks. The most notable exception is the work from Kapus and Cadar (2017) which proposed to randomly generate small programs and then perform differential testing, comparing in terms of outputs, function call sequences, and instruction line sequences, what observed during a native execution and what is observed during a symbolic exploration. However, as motivated in Section 2, this approach is most effective when considering symbolic frameworks, as modern concolic executors would by design *reproduce* the expected function calls, outputs, and instruction line sequences, thanks to the concrete execution used to *drive* the symbolic exploration along a path. Hence, when considering concolic frameworks, this approach can mainly find crashes during exploration but cannot identify inconsistent symbolic states arising from significant but not fatal implementation gaps. Moreover, this approach is not designed to test an engine with complex real-world applications, which can be quite limiting for a developer.

Our contribution. In this article, we investigate novel approaches that can help developers identify implementation gaps in concolic executors. Our interest in concolic executors arises from the recent efforts from the community into proposing more efficient and practical solutions, such as QSYM (Yun et al., 2018), SymCC (Poeplau and Francillon, 2020) SymQEMU (Poeplau and Francillon, 2021), Fuzzolic (Borzacchiello et al., 2021b), SymSan (Chen et al., 2022), and SymFusion (Coppa et al., 2022). We specifically target concolic execution because, as explained in Section 3, this approach can naturally embed several *consistency checks* thanks to the concrete execution performed in parallel with the symbolic evaluation of a program path. Our ideas can allow a developer to test a concolic framework over real-world programs, at each program operation, which is essential to identify unexpected implementation gaps that do not emerge when running on small and synthetic programs. In more detail, the contributions of this article are:

1. We describe the main steps carried out by several recent concolic frameworks during their analysis, pinpointing where implementation bugs may emerge. While our discussion does not provide a systematization that can be immediately generalized to all concolic engines, we believe that it can still provide some valuable insights;
2. We propose a set of novel and practical ideas on how to identify implementation gaps in modern concolic executors (Yun et al., 2018; Poeplau and Francillon, 2020; Borzacchiello et al., 2021b; Poeplau and Francillon, 2021; Chen et al., 2022), discussing the advantages and disadvantages of our proposed strategies;
3. We describe the implementation details of our strategies in the context of three concolic executors: SymCC, SymQEMU, and Fuzzolic. To favor reproducibility of our experiments and facilitate adoption of our techniques, we make available our code (Coppa, 2023a);
4. We report our own experience when applying these techniques to the three concolic frameworks for which we implemented our strategies. We experimentally evaluate their effectiveness in identifying actual bugs in the considered frameworks. Overall, our approach was able to discover more than 12 bugs across the considered concolic engines.

Structure of the article. Section 2 provides the background on several technical aspects related to concolic execution, helping the reader grasp the ideas presented in the following sections. Section 3 introduces our novel ideas, pinpointing their benefits and downsides. Section 4 describes how we implemented these ideas into three existing concolic frameworks. Section 5 reports the results of our experimental evaluation. Finally, Section 6 provides concluding remarks and insights for future research directions.

2. Background

2.1. Concolic execution

In this article, we present several strategies for testing implementations of concolic execution (Cadar and Sen, 2013b), one popular dynamic twist of symbolic execution (Baldoni et al., 2018). While the term is common to a large body of research works (Godefroid et al., 2008, 2012; Sen et al., 2005; Godefroid et al., 2005; Cadar et al., 2008b; Poeplau and Francillon, 2020; Yun et al., 2018; Borzacchiello et al., 2021b; Poeplau and Francillon, 2021; Chen et al., 2022; Coppa et al., 2022), it actually may refer to different flavors of the same essential idea: mixing symbolic and concrete execution. In particular, past works have looked at such mixing from different perspectives and with different trade-offs in mind, e.g., by targeting single functions or entire programs, by evaluating the program behavior through code instrumentation or by relaying on analysis of execution traces, using the concrete state to simplify non-linear constraints or exploit the native execution to skip the symbolic evaluation of complex sequences of code. Since considering the nuances of all the existing flavors is

```
  void foo() {
1.  uint a = get_input();
2.  uint b = get_input();
3.  if (a == 0xBAD)
4.    if (b == 0xCAFE)
5.      bug();
6.  return;
  }
```

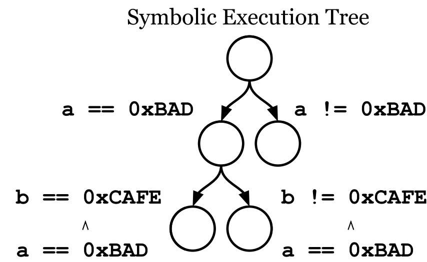

Fig. 1. A simple function used to explain concolic execution.

extremely hard, this article focuses on the flavor of concolic execution exploited by several recent works, including QSYM, Fuzzolic, SymCC, SymQEMU, SymFusion, and SymSan. To make clear how this flavor works, we present an example. From now on, when we use the term *concolic execution*, we always consider this specific flavor.

Example. We consider the simple function[1] `foo` from Fig. 1, which takes two inputs, `a` and `b`, respectively. Since concolic execution is a dynamic technique, meaning that we actually run the code under analysis, differently from traditional symbolic execution, we start by choosing an initial value for the two inputs and then run the code. For instance, let us pick `a = 0` and `b = 0`, which makes the native execution of `foo` follow the path along lines 1, 2, 3, and 6. While running the code, concolic execution needs to monitor the program behavior, tracking when and how the computations or branch decisions depend on the input data. To allow such inspection capabilities, a common practice is to exploit code instrumentation, either injected at compilation time (as done by, e.g., SymCC) or at running time (as done, e.g., by SymQEMU or Fuzzolic). For the remainder of this discussion, we assume that the code has been instrumented with one of these two approaches.

Thanks to the instrumentation, concolic execution can thus detect that `foo` at lines 1 and 2 is obtaining two new inputs, `a` and `b`, respectively. Since the goal of concolic execution is to understand how the program may behave when the input data changes, the technique in response generates two *symbolic* inputs, α_a and α_b, respectively. Hence, while `a` and `b` have a fixed value in the native execution, concolic execution assumes – within the symbolic execution carried out in parallel – that α_a and α_b can take any value admissible for their data type: in our example, the technique thus assumes that both α_a and α_b can take any value within the range $[0, 2^{32} - 1]$. However, as the execution proceeds along the path, the code will take specific branch directions, assuming after each decision that some conditions over the inputs are *true* for the current path. A concolic engine tracks such program decisions using the *path constraints* π. Initially, π is equal to *true* since no decision has been made yet by the function.

At line 3, the code evaluates the branch condition `a == 0xBAD`. In the native execution, the condition is *false* since `a = 0`. On the other hand, in the symbolic execution, this condition may be *true* or *false* depending on the specific value assigned to α_a. Since the current input already provides an assignment to α_a that can make `foo` take the *false* direction, concolic execution focuses on the *true* direction. In particular, it builds the symbolic expression $\alpha_a == \texttt{0xBAD}$, puts it in conjunction with the current π (generating $\alpha_a == \texttt{0xBAD} \land true$) and then queries the SMT solver to possibly obtain a satisfying assignment for such expression. Assuming that a state-of-the-art solver may return the assignment $\alpha_a = \texttt{0xBAD}$, the engine builds a new set of input values `a = 0xBAD` and `b = 0`, which is stored in a queue for later use. After performing the branch query, the engine updates π to $\alpha_a \neq \texttt{0xBAD}$ since the native execution will go on under this assumption. The native execution will then reach the end of the function, terminating the concolic execution for the current set of inputs.

The engine now picks another set of inputs from the queue, starting a new native execution with `a = 0xBAD` and `b = 0`. The execution now follows the path along lines 1, 2, 3, 4, and 6. At line 3, the code takes the *true* direction. Since the initial input values have already visited the *false* direction of line 3, no query is submitted to the SMT solver. Nonetheless, π is updated to $\alpha_a == \texttt{0xBAD}$ to keep the symbolic execution in sync with the decisions taken by the path. At line 4, since the code is taking the *false* direction, the concolic engine generates the symbolic expression $\alpha_b == \texttt{0xCAFE}$ related to the *untaken* direction, puts it in logical conjunction with the previous path constraints, building the expression $\alpha_b == \texttt{0xCAFE} \land \alpha_a == \texttt{0xBAD}$, and then queries the SMT solver to possibly obtain a satisfying assignment. Assuming the solver returns `a = 0xBAD` and `b = 0xCAFE`, a new set of values is added to the input queue. The native execution now reaches the end of the function. Hence, again the engine picks from the input queue, starting a new concolic run using `a = 0xBAD` and `b = 0xCAFE`. The native execution follows the path along lines 1, 2, 3, 4, and 5, reaching the interesting function `bug`. The engine can thus report the current set of inputs to the user, allowing her to reproduce the path even in traditional native execution.

Comparison with symbolic execution. When comparing concolic execution with the original technique, the most important aspect to consider is that concolic execution analyzes the program *one path at a time*, performing symbolic execution along the path taken by the native execution. Conversely, traditional symbolic execution typically does not concretely run the original program and it is often used to explore several paths in parallel at the same time. A crucial benefit of the concolic approach is that the engine can avoid to explicitly keep track of the program's concrete state (as it can be retrieved from the native execution), possibly significantly reducing the analysis overhead in the presence of complex concrete computations. Moreover, it can rely on the concrete state to evaluate how to go on with the analysis even when the SMT solver cannot reason over a query or when a piece of code is not symbolically tracked by the engine.

2.2. Causes for implementation gaps

After presenting the main ideas behind concolic execution, we can now identify more precisely the essential steps carried our by recent concolic execution engines when analyzing a program. Fig. 2 attempts to provide a succinct visual overview of these analysis steps, which we now review in more detail, while also pinpointing possible sources for implementation gaps[2]: Fig. 2 depicts the main steps performed by a concolic executor during its analysis along a path. We now review these steps, pinpointing possible implementation gaps[3]:

[1] While our example is based on a single function, the concolic frameworks considered in this article target entire programs. Nonetheless, the main ideas remain the same.

[2] The goal of this article is not to exhaustively present all possible causes for implementation gaps but only to provide insights valuable for subsequent sections.

[3] The goal of this article is not to exhaustively present all possible causes for implementation gaps but only to provide insights valuable for subsequent sections.

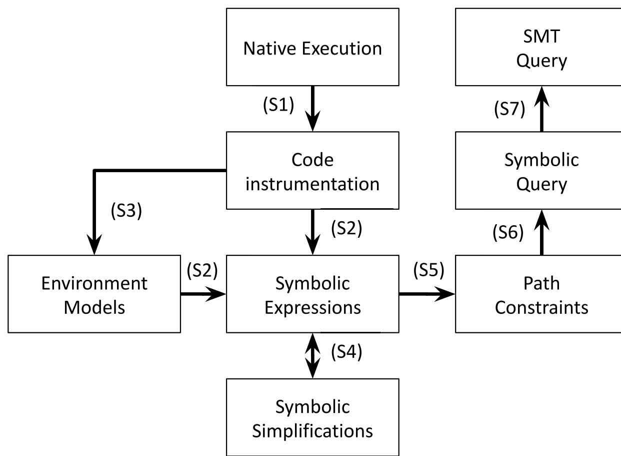

Fig. 2. Analysis steps performed by several recent concolic frameworks (Poeplau and Francillon, 2020; Borzacchiello et al., 2021b; Poeplau and Francillon, 2021; Yun et al., 2018).

S1 The native execution is often tracked through code instrumentation. The instrumentation may decide to directly build the symbolic expressions (S2) based on the current operation performed by the program or skip the analysis over a piece of code and use instead an environment *model* (S3) to keep the symbolic state consistent.
Implementation gap #1: a program operation semantics is not tracked accurately by the instrumentation, e.g., an addition is tracked as a subtraction.
Implementation gap #2: the framework ignores the side effects of a piece of code whose analysis is skipped, e.g., the effects of the system call `fstatfs` are ignored, failing to execute the related model.

S2 Symbolic expressions are built by calling primitives from an expression *builder*. These expressions can then be simplified (S4), used to update the path constraints (S5), and generate symbolic queries (S6).
Implementation gap #3: If the builder is implemented incorrectly, then the resulting symbolic expressions will be inaccurate. For instance, the builder may expose a primitive for the arithmetic negation (`-x`) but then the resulting expression may instead flip the value bits (`~x`).

S3 When the framework skips the analysis over a piece of code, it should devise an environment *model*, i.e., a hand-written function that replicates the side-effects of the skipped code, to still keep consistent the symbolic state.
Implementation gap #4: The framework devises a model that is inaccurate, e.g., it wrongly handles the semantics of the system call `mmap` omitting to update some symbolic expressions stored in the memory.

S4 Symbolic expressions could be often simplified to make them more compact and easier to process both for the developer (during debugging sessions) and for the SMT solver.
Implementation gap #5: One simplification may generate an expression that is not semantically equivalent to the original expression. For instance, the concatenation of two variables, such as `a << (sizeof(b) * 8) + b`, could be rewritten in the wrong way, such as `a | b` (where `|` is the bitwise OR operator) instead of `Concat(a, b)`.

S5 When reaching a branch, the engine should recover the symbolic expression e representing the branch condition and the taken direction in the path. If the branch is taken, e should be added to the path constraints. Otherwise, when the branch is not taken, $\neg e$ should be added to the path constraints. This step is in charge of handling this logic by correctly invoking the path constraint manager.
Implementation gap #6: The path constraints may be incorrectly updated, making infeasible all subsequent queries. For instance, in the presence of a condition that is trivially *true* (e.g., `2 > 1` simplified into *true* during S4), the executor may negate it (by mistake), adding a condition $false$ to the path constraints, making them unsatisfiable.

S6 This step is the continuation of S6 and is in charge of building new symbolic queries to generate alternative inputs with the solver (S7).
Implementation gap #7: The symbolic expression e for a branch may not be negated as expected, leading to the generation of useless inputs.

S7 Symbolic queries generated by S6 must be translated into a solver-specific language to allow the SMT solver to reason over them.
Implementation gap #8: The translation into the solver-specific expression representation is wrongly implemented. For instance, the arithmetic negation (`-x`) may be translated into the Z3 expression representation using `Z3_mk_bvnot` (logical negation) instead of `Z3_mk_bvneg` (arithmetic negation).

The consequences of these implementation gaps can be quite different:

- Some queries submitted to the solver could be wrong, failing to generate valuable inputs or generating inputs that would not reproduce the expected path;
- The path constraints may at a certain point contain wrong and unfeasible conditions, making *all* the subsequent queries infeasible;
- In some cases, the queries could be *slightly* wrong, containing conditions that are very complex to reason on for the SMT solver, resulting in high solving times and thus a large number of solver timeouts.

2.3. Existing approaches for testing symbolic frameworks

The research community, to the best of our knowledge, has not thoroughly investigated methodologies that can support a developer when testing symbolic or concolic execution frameworks. The most notable exception is the work from Kapus and Cadar (2017) which proposed to:

- generate simple and random programs with CSmith (Poeplau and Francillon, 2020);
- perform differential testing between the native execution and the symbolic exploration.

The approach aims at detecting the following problems:

1. *Crashes during the symbolic exploration*: this is the most straightforward strategy to suggest and it is valuable even for concolic executors. However, several implementation bugs may never generate a fatal crash and thus go unnoticed. An essential contribution of this article is the proposal of several strategies to detect inconsistencies during concolic execution and then artificially generate a crash, which can then be externally tracked and investigated;
2. *Inconsistent program output*: given a randomly generated program, it checks whether the output, artificially generated as the checksum of several global variables, is equal to the value obtained during a native execution. Unfortunately, in the case of concolic execution, the approach seems to consider the concrete values associated with the output. However, recent concolic executors are typically designed to keep the native execution consistent and only add instrumentation code to carry out the symbolic execution, making unlikely the generation of inconsistent output by the instrumented program;

3. *Divergent instruction sequence*: the idea is to constrain the symbolic variables to a single fixed value and then see if the framework is covering the expected instruction sequence consistently with the native execution. This strategy is not effective for a concolic framework as the exploration is by design driven by a native execution, leading to the expected instruction sequence;
4. *Divergent function sequence*: the idea is to make the framework generate an input related to an alternative branch direction, run the program over the input, and then see if the program is reproducing the expected function sequence up to the branch. While this is relevant even for concolic frameworks, it is not straightforward to implement: the original program may have been optimized in different ways (e.g., different inline optimizations), possibly generating a slightly different function sequence than what observed during the concolic execution (which may require to recompile the program with a custom compiler pass). Hence, this strategy can be extremely valuable but require some adjustments to be more *practical* for a developer (see Section 3.3.2).

One essential aspect of this approach is the choice of using programs randomly generated with CSmith, which brings its benefits but also downsides. From one side, it helps find implementation gaps due to *unexpected* code patterns, helping to cover even uncommon corner cases. Also, such programs do not come with non-deterministic aspects and be easily reduced to minimal test cases, making easier the life of a developer during debugging. On the other side, however, programs generated by CSmith hardly interact with the operating system, by calling library functions or using system calls, and thus cannot always reproduce complex behaviors emerging in real-world programs, which, however, may be of interest for a developer

Overall, the following differences with our work can be identified:

- *Testing using any arbitrary program*: our approach aims at providing testing strategies that can be applied when considering any arbitrary program. This allows developers to exploit our techniques even on, e.g., a specific program target of their choice. Hence, they can get confidence (even if no guarantees can be provided) that the concolic execution is *tested* on that specific target;
- *Testing granularity*: our approach aims at providing fine-grained checks, allowing developers to test, e.g., any new symbolic expression built by an engine. On the other hand, the approach from Kapus and Cadar (2017) is designed to perform more coarse-grained checks, validating the execution at specific points in time. A natural downside of our design choice is that our checks would likely bring more overhead. Nonetheless, the low granularity of the checks may help also raise an alert near the root cause of the problem;
- *Self-contained testing*: The work from Kapus and Cadar (2017) proposes a strategy based on differential testing, which is a quite powerful approach. However, it can be quite *impractical* from a developer's point of view. Indeed, the approach performs checks that rely on a comparison with the original program. However, concolic executors work on an instrumented version of the program, making it not always trivial a mapping with the original uninstrumented code. Our approach does not require to make a comparison with something that is *outside* the concolic executor. Indeed, as explained in Section 3, we rely on an *internal* comparison between the native execution and the symbolic execution, both by design carried out in parallel by the same engine. Although our choice makes our approach more *self-contained*, it also relies on the correctness of the instrumented native execution.

As a last remark, we believe that our proposal is complementary to the solution proposed by Kapus and Cadar (2017) since both approaches come with their own benefits and downsides, without claiming the superiority of one over the other.

2.4. Existing approaches for testing program analysis frameworks

Several previous works (Cadar and Donaldson, 2016; Cuoq et al., 2012; Daniel et al., 2007; Roy and Cordy, 2009; Wu et al., 2013; Chen et al., 2020; Yang et al., 2011) have tackled the problem of testing software analysis frameworks.

Cadar and Donaldson (2016) highlight the importance of cross-checking program analyzers to find inconsistencies, suggesting to use program transformations on real-world programs to generate targets with known bugs. They also propose to make program generators more tunable in order to ease the testing of specific analyzers. Our strategies are agnostic to the target under analysis and thus their suggestions could be applicable even for our work.

Cuoq et al. (2012) report their experience when using CSmith to test the static analyzer Frama-C. In particular, they build several oracles for detecting incorrect analysis results on the generated programs: e.g., they evaluate whether a *sliced* version of the program, emitted by Frama-C, computes at running time a different checksum with respect to an execution of the original program. Unfortunately, their oracles are tailored to Frama-C and cannot be easily reused in the context of symbolic execution.

Daniel et al. (2007) focus on the testing of refactoring engines by proposing ASTGen, a library that can allow developers to produce Abstract Syntax Trees (ASTs) with structural properties relevant for the refactoring functionalities. For example, ASTGen can generate programs containing variables with potential name clashes to validate a *rename* variable refactoring feature. The generation of programs with custom ASTs could be relevant for symbolic execution since it may help test the generation of specific symbolic expressions (e.g., expressions with special operators). We see this direction as an interesting future work.

Roy and Cordy (2009) target instead the testing of software clone detectors. Their goal to is to evaluate the precision and recall of such tools by exploiting a mutation-based approach able to support different types of fine-grained copy, paste, and modify code clone operations. Similarly to custom ASTs, we believe that custom transformations on the target program could be valuable for testing specific features of a symbolic framework.

Wu et al. (2013) consider the testing of alias analysis implementations. Their approach instruments programs to track at a running time when pointers have the same value, i.e., they are an *alias*, and emits an alarm when alias analysis results contradict what has been observed during the execution. While their idea is not applicable to symbolic execution, it shares some traits with the spirit of our work.

One research direction that has seen a large number of works (Chen et al., 2020; Yang et al., 2011) is related to the testing of compilers, which internally integrate several program analyses. A key idea behind a large chunk of such works is to emit source programs that make the compiler misbehave. These programs can be built using: (a) grammar-directed approaches, that emit code by, e.g., exploiting a context-free grammar of the source language; (b) grammar-aided techniques, that, e.g., generate programs exploiting the language grammar but starting from, e.g., template-like code fragments; (c) random-based generation solutions, that randomly compose excerpts of code; (d) mutation strategies, that modify an initial program using several mutations with the goal of either generate equivalent but different variants of the same program or to be non-semantics-preserving but still trying to avoid undefined behaviors. To evaluate whether a compiler behaves correctly, most of these works rely on test oracles to perform more specialized checks.

Overall these works do not specifically target symbolic (or concolic) execution and thus are not immediately comparable with our approach. Nonetheless, they confirm the crucial need for methodologies able to support developers during the testing of program analysis frameworks.

3. Approach

In this section, we describe several testing strategies that may help a developer to identify the implementation gaps affecting the analysis steps exemplified in Section 2.2. A practical observation that motivates the design of our strategies is that it may be often infeasible, or at least overly expensive, to check the *correctness* of the steps executed by a concolic executor. Indeed, state-of-the-art concolic frameworks are not implemented following approaches amenable to existing formal approaches. Hence, we propose instead to check at least the *consistency* of the analysis steps by taking into account the information available from the concrete state, i.e., the concrete execution carried out in parallel with the symbolic evaluation. While this intuition may seem naive, it can be quite effective in practice and is not yet adopted by most engines.

3.1. Notation

To ease our discussion, we define the following terminology:

- An object o is any aspect of the program state that is symbolically tracked by the concolic executor. Each object will have a size measured in bytes. For instance, an object could be one architecture-specific register, such as `RAX` on `x86_64`, whose size would be 8, or a 32-bit C variable `x`, whose size is 4, or a memory byte, such as the 8-bit data available at address `0x000005555`, whose size would be 1;
- Object I is the program input. We treat it as a special *abstract* object. In practice, it represents bytes read from, e.g., a file or a socket;
- P is a program point, i.e., one instruction during the native execution of the program over I;
- $native(o)$ is the native (concrete) value of the object o. For instance, $native(I)$ is the sequence of concrete bytes composing the input used when running the program during the concolic execution. E.g., a 4-byte input I may have $native(I) \mapsto$ {`0xA`, `0xB`, `0xC`, `0xD`}. The subscript notation can be used to refer to one specific concrete byte within the object, e.g., $native(I)[0] \mapsto$ `0xA`;
- $symbolic(o)$ is the symbolic expression associated with the object o. The input I by design involves only *pure* symbolic data, e.g., for a 4-byte input, we may have $symbolic(I) \mapsto \{\alpha_0, \alpha_1, \alpha_2, \alpha_3\}$ where α_i represents the ith symbolic byte from I. Other objects may instead involve *pure* symbolic data, e.g., $symbolic(\texttt{RAX})[0] \mapsto \alpha_0$ when the first byte of `RAX` is a mere copy of the first symbolic input byte, or a *derived* symbolic expression, e.g., $symbolic(\texttt{RAX})[0] \mapsto \alpha_0 + 1$ when the first byte of `RAX` is a computation over the first symbolic input byte, or even concrete data, e.g., $symbolic(\texttt{RAX}) \mapsto$ `0xB` when `RAX` does not depend from the input and its value is equal to the concrete value `0xB`. When $symbolic(o)$ is concrete, then most modern concolic executors discard the associated expression since $symbolic(o) = native(o)$ and they set $symbolic(o)$ to a sentinel empty value, e.g., `NULL`;
- $evaluate(e, I)$ is a function that replaces any α_i within the symbolic expression e with the related concrete value from $native(I)$ and then computes the resulting concrete value c for the expression e through constant folding. For instance, $evaluate(\alpha_0 + 1, I) \mapsto 1$ when $native(I)[0] \mapsto$ {`0x0`}, since `0x0 + 1` $\mapsto$ `0x1`. For the sake of simplicity, we assume that $evaluate$ returns `0x1` for $true$ conditions and `0x0` for $false$ conditions. This evaluator could be implemented in two main ways:
  1. *Internal evaluator.* The concolic engine devises an evaluator for its symbolic expression representation. Notice that some concolic executors may already implement such evaluator for their own functionalities, e.g., to forcefully concretize an object to limit path explosion. However, the evaluator may by itself introduce new implementation gaps when developed incorrectly;
  2. *Solver-based evaluator.* The concolic engine may delegate the evaluation to the SMT solver. Indeed, solvers typically allow an engine to define *assignments* for the symbolic pure data and then evaluate an expression to obtain the resulting concrete value. This approach is simpler from the implementation point of view and we favor it in this article.

  Notice that the running time cost for the evaluation of an expression is expected to be linear with respect to the number of operators involved in the expression e since the evaluation mainly requires a bottom-up visit of the expression;
- $o_{res} = op(o_1, \ldots, o_i, \ldots)$ is an operation performed by the program, where op is an operator (e.g., add, $store$, cmp_eq, etc.), o_i is the object used as ith operand by the operator (e.g., register `RAX` in `x86_64`), and o_{res} is the object where the program will store the result of the operation (e.g., register `RCX` in `x86_64`);
- π_P is the set of path constraints that the concolic executor has collected when analyzing the current path up to a specific program point P.

3.2. Checking the consistency of the symbolic expressions

Our first kind of testing strategies is aimed at detecting inconsistent symbolic expressions. We devise it into two variants.

3.2.1. Consistency strategy CHKEXPR

Intuition. The goal of this strategy is to check whether the symbolic expression for an object o is consistent with its native value. In other words, whenever the concolic execution performs an operation:

$$o_{res} = op(o_1, \ldots, o_i, \ldots)$$

it should check, before executing the operation, whether:

$$\forall i : evaluate(symbolic(o_i), I) \stackrel{?}{=} native(o_i)$$

and, after executing the operation, whether:

$$evaluate(symbolic(o_{res}), I) \stackrel{?}{=} native(o_{res})$$

These consistency checks can be done while performing the original concolic exploration over the input I.

Example. For instance, let us consider the C instruction:

$$\mathrm{x} = \mathrm{x} + 1$$

where:

$$symbolic(\mathrm{x}) \mapsto \alpha_0$$

and:

$$native(I) \mapsto \{\texttt{0x0}, \texttt{0x0}\}$$

Then, before the addition, this strategy would check that:

$$evaluate(symbolic(\mathrm{x}), I) = evaluate(\alpha_0, I) \stackrel{?}{=} native(\mathrm{x}) = 0$$

and, after the computation, it would check that:

$$evaluate(symbolic(\mathrm{x}), I) = evaluate(\alpha_0 + 1, I) \stackrel{?}{=} native(\mathrm{x}) = 1$$

Discussion. This strategy can catch implementation gaps in S1, S2, S3, and S4, since an inconsistency may arise due to incorrect code instrumentation (S1), incorrect implementation of the builder (S2), incorrect environment modeling (S3), and incorrect simplifications (S4). Whenever a solver-based evaluator is used, then this strategy can also catch problems due to improper translation into the solver-specific expression representation (S7).

One notable downside of this strategy is that it may miss implementation gaps that are not leading to inconsistencies for the current $native(I)$. For instance, in our example, if the engine, after the addition,

builds the expression α_1+1 instead of α_0+1, then it would not detect the inconsistency since $evaluate(\alpha_0+1, I) = evaluate(\alpha_1+1, I)$ when using $native(I) \mapsto \{\texttt{0x0}, \texttt{0x0}\}$. This scenario is common when considering branch conditions whose values can be either $true$ (0x1) or $false$ (0x0), making it unlikely to detect inconsistencies since most input assignments may lead to the same branch evaluation.

3.2.2. Consistency strategy FUZEXPR

Intuition. The goal of this strategy is to improve the capabilities of CHKEXPR at detecting implementation gaps. In particular, given an expression for an object o at a program point P, it will ask the SMT solver to find an I' able to satisfy π_P and such that:

$$evaluate(symbolic(o), I) \neq evaluate(symbolic(o), I')$$

In other words, the engine aims at obtaining an input I' able to generate a different concrete value of the object o when running the program using I'. Notice that the solver may fail to generate I' for two main reasons: (a) given the current path constraints π_P, it is not possible to satisfy such query, or (b) the solver cannot answer the query within a reasonable amount of time.

Whenever the solver can generate an input I', the concolic executor can perform a new concolic execution based on I' with strategy CHKEXPR enabled to check for inconsistencies over o. Since performing a new full exploration with strategy CHKEXPR enabled on all operations could be overly expensive, the engine may instead opt to only use strategy CHKEXPR at program point P, skipping checks on previous program points and aborting its execution after reaching P. Moreover, the engine has to choose how many times (k) performs FUZEXPR checks for the same object o at a program point P. Indeed, the solver may be able to generate several alternative inputs, each one requiring a different concolic execution to be checked.

Example. For instance, let us consider the C instruction at a program point P:

$$\texttt{x} = \texttt{x} + 1$$

where:

$$symbolic(\texttt{x}) \mapsto \alpha_0$$

and:

$$native(I) \mapsto \{\texttt{0x0}, \texttt{0x0}\}$$

If, after the computation, the engine generates:

$$symbolic(\texttt{x}) \mapsto \alpha_1 + 1$$

then strategy CHKEXPR would not identify the implementation gap. However, when using strategy FUZEXPR, the engine may ask the solver whether the following query is feasible:

$$\alpha_1 + 1 \neq 1$$

Assuming the solver returns input I':

$$native(I') \mapsto \{\texttt{0x0}, \texttt{0x1}\}$$

Then, when performing a new concolic execution over I', the engine can identify the inconsistency at program point P with strategy CHKEXPR.

Discussion. Strategy FUZEXPR is potentially more powerful than strategy CHKEXPR but it is also more expensive as it requires to: (a) perform a query with the SMT solver, and (b) perform a new concolic exploration over I'. To cope with (a), we suggest using a small solving timeout, to avoid waiting too much time for a query response, or relying on approximate solvers, such as FuzzySAT (Borzacchiello et al., 2021a). Both of these approaches limit the running time cost but also reduce the capabilities of strategy FUZEXPR at identifying implementation gaps, hence they must be seen as a trade-off. To further mitigate (b), an engine may decide to FUZEXPR only within a limited set of program points, however, accepting again to possibly miss some implementation gaps.

3.3. Checking the consistency of the path constraints

Our second kind of testing strategies aims at identifying inconsistent path constraints. Similar to Section 3.2, we propose it into two flavors.

3.3.1. Consistency strategy CHKPC

Intuition. The goal of this strategy is to check the consistency of the path constraints with respect to the native execution. During concolic execution, the path constraints by design should be satisfied by $native(I)$ since I is driving the path exploration. Hence, this strategy checks that:

$$\forall P : evaluate(\pi_P, I) \overset{?}{\neq} \texttt{0x0}$$

In practice, any time π is updated, the strategy should check its consistency.

Example. Let us suppose that there is a bug that makes π_P equal to:

$$\pi = (\alpha_0/10 > 0) \wedge (\alpha_0 < 0)$$

when:

$$native(I) \mapsto \{\texttt{0x0}, \texttt{0x1}\}$$

Then, using this strategy, the concolic executor computes:

$$evaluate(\pi, I) = \texttt{0x0}$$

thus detecting that there is an inconsistency.

Discussion. While several symbolic executors may already check whether π_P is satisfiable when running in debugging mode, we underline that this kind of check is not the correct one in the context of concolic execution and it is not equivalent (or more general) than our strategy. Indeed, checking that the path constraints are feasible means that the solver can identify one assignment able to satisfy them. However, the path constraints, due to implementation gaps, may not represent faithfully the current path but then still admit one feasible solution. For instance, suppose that $native(I) \mapsto \{\texttt{0x0}\}$ and the concolic engine has built the wrong path constraint $\alpha_0 \neq 0$ where $symbolic(I)[0] \mapsto \alpha_0$ then the solver can find an assignment that satisfies the path constraints, missing the implementation gap. On the other hand, when checking $evaluate(\alpha_0 \neq 0, I)$, our strategy can detect that the current path constraints are not satisfied by the current input, pinpointing that they are inconsistent with the native execution. Moreover, our evaluation does not require heavyweight reasoning in the solver, making our strategy often faster than an approach performing a satisfiable check query.

3.3.2. Consistency strategy CHKINP

Intuition. Concolic executors are often used to generate program inputs that would make the program reach specific program points, e.g., generate an input that makes the program under analysis visit the $false$ direction of a given branch. Strategy CHKINP is aimed at verifying whether an input I', generated when aiming at reaching a program point P, actually reaches P during the concolic execution over I'. To implement this strategy, the concolic executor has to define a mechanism able to assign a unique identifier to each program point. In practice, we observed that it in several cases it could be enough to hash the addresses of the executed instructions under the assumption of a fully deterministic native run.[4]

Example. Suppose that the program has the following code:

```
if(x > 0){/ * P1 * /}else{/ * P2 * /}
```

[4] We thus suggest disabling address randomization and limiting other possible sources of non-determinism. Nonetheless, in general, fully avoiding or controlling non-determinism in arbitrary programs can be challenging.

and:

$symbolic(\mathrm{x}) \mapsto \alpha_0$

Then, when:

$native(I) \mapsto \{\mathtt{0x1}, \mathtt{0x0}\}$

the native execution would reach point P1 and the concolic executor may generate:

$native(I') \mapsto \{\mathtt{0x0}, \mathtt{0x0}\}$

when aiming at reaching point P2. Then strategy ChkInp should check whether a new concolic execution over I' is indeed reaching point P2. If this is not happening, then the concolic executor is experiencing path divergence, which may be either caused by a bug or due to a non-deterministic factor. Even in the latter case, the developer is likely interested in identifying such a problem.

Discussion. This strategy is quite natural and it is likely not novel. Indeed, Kapus and Cadar (2017) already proposed this idea, however, we suggest to check the expected path within a new concolic exploration and not within a traditional native execution. While our choice may hide some bugs, when concolic execution is not following the path of the native execution, it nonetheless makes this strategy practical to implement for developers since they do not need to reason over a different execution environment, e.g., outside the DBT, or over a different program, i.e., the uninstrumented program, making easier to identify P2 when checking I'. Moreover, while our suggestion may seem trivial, we remark that most modern concolic executors do not ship with a mechanism to check such consistency. We thus included it in this article because any developer should be aware of this strategy. Unfortunately, this strategy is not *debug-friendly*: it cannot detect by itself at which program point P′ the native path is diverging from the expected path. Indeed, to finely identify the divergent program point, the concolic executor should finely compare the path over I and the path over I'. This is technically possible but requires additional effort, e.g., dumping a fine-grained execution trace for a path. Another important consideration about ChkInp is related to its cost. Similarly to FuzExpr, it requires to perform a new concolic run for each check. However, differently from FuzExpr, the number of branches for which a concolic executor is likely able to generate an alternative input is expected to be limited given the complexity of real-world programs.

3.4. Checking the consistency of the expression simplifications

Implementing expression optimizations is not easy and thus our last kind of testing strategies aims at identifying inconsistencies due to incorrect expression simplifications. As for previous kinds, we design this strategy into two variants.

3.4.1. Consistency strategy evopt

Intuition. A concolic executor may perform several simplifications over a symbolic expression e, generating an *optimized* expression e'. The aim of this strategy is to check whether e' is consistent with e, i.e.,:

$$evaluate(e, I) \stackrel{?}{=} evaluate(e', I)$$

Example. Let us consider the expression e:

$((\alpha_1 \ll 8) | \alpha_0) \wedge \mathtt{0xFF00}$

and that a concolic executor may simplify it into the expression e':

$\alpha_1 \ll 8$

since the bitmask 0xFF00 discards the 8 bits of α_0. When:

$native(I) \mapsto \{\mathtt{0xCA}, \mathtt{0xFE}\}$

This strategy can check that:

$$evaluate(((\alpha_1 \ll 8)|\alpha_0) \wedge \mathtt{0xFF00}, I) = \mathtt{0xFE00} = evaluate(\alpha_1 \ll 8, I)$$

Discussion. This strategy is equivalent to ChkExpr. However, we propose it for two main reasons: (a) a developer may decide to skip ChkExpr checks on most operations to limit the slowdown during the concolic exploration, while still wanting to check all simplifications and (b) ChkExpr checks could be intended for checking *new* expressions rather than for checking *rewritten* expressions, making ChkExpr not effective in the latter case.

Similarly to strategy ChkExpr, there are several improper simplifications that may not lead to inconsistencies when evaluated in the context of a specific native(I). For instance, in the previous example, when considering an incorrectly simplified expression e' equal to $\alpha_0 \ll 8$ and $native(I) \mapsto \{\mathtt{0x00}, \mathtt{0x00}\}$ then the strategy would fail to detect an inconsistency since:

$$evaluate(((\alpha_1 \ll 8)|\alpha_0) \wedge \mathtt{0xFF00}, I) = \mathtt{0x00} = evaluate(\alpha_0 \ll 8, I)$$

3.4.2. Consistency strategy smtopt

Intuition. This strategy is a stronger version of the previous one. In particular, the concolic executor can use the SMT solver to verify that the query:

$e \neq e'$

is not feasible, i.e., the two expressions are semantically equivalent.

Example. Let us consider again the expression e:

$((\alpha_1 \ll 8)|\alpha_0) \wedge \mathtt{0xFF00}$

and that a concolic executor may wrongly simplify it into the expression e':

$\alpha_0 \ll 8$

Then, this strategy can check that:

$((\alpha_1 \ll 8)|\alpha_0) \wedge \mathtt{0xFF00} \neq \alpha_0 \ll 8$

is satisfiable, obtaining from the solver the following assignments:

$native(I') \mapsto \{\mathtt{0xCA}, \mathtt{0xFE}\}$

which can indeed prove the inconsistency since:

$$evaluate(((\alpha_1 \ll 8)|\alpha_0) \wedge \mathtt{0xFF00}, I') = \mathtt{0xFE00} \neq \mathtt{0xCA00} = evaluate(\alpha_0 \ll 8, I')$$

Discussion. While this strategy is more powerful than EvOpt, it is also more expensive and not always actionable. Indeed, the SMT solver may take a very long time to answer the query, possibly forcing the concolic executor to give up before obtaining a response. This may not be uncommon when considering expressions involving non-linear constraints, such as division operations, where the solver may struggle at reasoning over. The partial mitigation, in exchange for accuracy, could be to use a small solving timeout for SmtOpt and then, in case of timeout, rely on strategy EvOpt.

3.5. Summary

Table 1 summarizes the main ideas behind our testing strategies, reporting for each of them the expected number of checks. ChkExpr evaluates any expression built by the engine in the context of the current input and then checks whether the computed value is consistent with the concrete value kept by the native execution. Since ChkExpr checks the consistency only with respect to the current input, we proposed FuzExpr to specifically generate inputs that induce alternative values for an expression. These alternative values are generated using a solver and the user can control their maximum number through a threshold k. Each alternative value will require to perform another execution, exploiting ChkExpr to check the expression consistency. ChkPC and ChkInp focus on the branch conditions, where the former checks the satisfiability of π, while the latter checks whether the input generated for the alternative direction of a branch replays the expected

Table 1
Summary of the proposed testing strategies.

Strategy	Description	# checks
ChkExpr	Evaluate an expression w.r.t. the current input and check consistency against its native value	One for each expression
FuzExpr	Identify with the solver inputs that induce different values for an expr. and then exploit ChkExpr to check them.	Up to k concolic exec. for each expr., where k is a user-defined threshold
ChkPC	Evaluate π w.r.t. the current input to check its satisfiability	One for each branch
ChkInp	Check whether a generated input leads to the expected path	One concrete exec. for each input generated along a path
EvOpt	Evaluate an expression w.r.t. the current input before and after optimizing it	One for each expr. optimization
SmtOpt	Check using the solver whether an expr. may assume different values before and after optimizing it	One for each expr. optimization

subpath. Finally, EvOpt and SmtOpt both test the expression optimizations, checking whether the optimized expression is *different* from the unoptimized one, where EvOpt performs an evaluation in the context of the current input while SmtOpt relies on the SMT solver.

Relationship among strategies. Strategy ChkPC is a more powerful variant of ChkExpr since it attempts to validate the consistency of an expression under different inputs. However, ChkPC could be significantly more expensive as it relies on the SMT solver and requires to perform additional executions. Strategy ChkPC and ChkInp both aims at checking the consistency of the path constraints, where ChkPC focuses on the current path while ChkInp validates the predicted path constraints for the alternative paths. To generate inputs for the alternative paths, ChkInp relies on the SMT solver. Strategy EvOpt is a weaker version of SmtOpt that, however, does not make use of the solver and thus should be more efficient.

Exploiting the strategies in the context of symbolic execution. Strategies ChkExpr, FuzExpr, ChkPC, and EvOpt are concolic-specific since they exploit the native state to perform the consistency checks, thus are not immediately reusable in symbolic engines. They could be adapted even for other flavors of concolic executions as they mainly require to implement the *evaluate* and *native* functions. The other two strategies, ChkInp and SmtOpt, can instead be used even in symbolic execution.

False positives and false negatives. When our strategies identify an inconsistency, then this means that there is indeed a bug in the concolic framework, i.e., they do not generate false positives. There are only two *worthy* considerations to keep into account. First, our techniques assume that the native execution carried out by the concolic engine is consistent with an execution of the original program. Developers that may want to test this assumption should rely on the approach from Kapus and Cadar (2017). Second, in the presence of non-determinism factors in the program, our techniques cannot distinguish between a bug in the framework and a non-determinism aspect of the execution. Nonetheless, they could be valuable to detect such factors (indeed, they correctly identify inconsistencies across different runs).

Our strategies can have false negatives. First, the strategies FuzExpr, ChkInp, and SmtOpt rely on the help of the solver. However, the solver may not answer within a limited amount of time, preventing them from possibly revealing some inconsistencies. Second, the other strategies check the consistency in the context of the current input, possibly failing to detect inconsistencies emerging only when considering other inputs. Overall, as explained at the beginning of this section, our strategies cannot prove the correctness of the implementation but only aims at validating its consistency.

4. Implementation details

We implemented our strategies in three state-of-the-art concolic engines:

- SymCC (Poeplau and Francillon, 2020): This concolic executor performs source code instrumentation using an LLVM pass, injecting calls to a *runtime* that is in charge of building the symbolic expressions and reasoning over them using an SMT solver. We considered the *simple* backend for the runtime since it is the one suggested by the authors for debugging purposes. This backend directly builds Z3 expressions and relies on the SMT solver for expression optimizations. Our changes[5] involved ∼1000 C++ LOC;
- SymQEMU (Poeplau and Francillon, 2021): Differently from SymCC, this concolic executor performs binary code instrumentation exploiting the QEMU JIT engine, injecting calls to a *runtime* that is in charge of building the symbolic expressions and reasoning over them using an SMT solver. SymQEMU has been proposed by the same authors of SymCC and, to make our evaluation more interesting, we considered the *QSYM* backend for its runtime: this backend is heavily based on a subset of QSYM (Yun et al., 2018), an existing concolic executor that is now unmaintained. Differently from the simple backend, the QSYM backend builds the symbolic expressions using its own representation and adopts several custom optimization strategies. Nonetheless, it still relies on the SMT solver Z3. Our changes[6] involved ∼1500 C++ LOC;
- Fuzzolic (Borzacchiello et al., 2021b): Similarly to SymQEMU, this concolic executor performs binary code instrumentation through QEMU. However, differently from SymQEMU, the runtime runs in a different process from the native executor, requiring a custom communication protocol among the two components. Within the backend, Fuzzolic directly builds Z3 expressions but then applies several custom optimizations over such expressions. Our changes[7] involved ∼2000 C LOC.

Besides the changes required for integrating our strategies into the concolic engines, we also developed several utilities and a full working artifact[8] based on Docker to help developers perform the experiments described in the next section, possibly favoring reproduction of our results.

5. Experimental evaluation

In this section, we experimentally evaluate the effectiveness of our strategies when exploited in the context of three state-of-the-art concolic executors.

[5] Our fork can be found at https://github.com/ercoppa/symcc-debug-ce.
[6] Our forks can be found at https://github.com/ercoppa/symqemu-debug-ce and https://github.com/ercoppa/qsym-debug-ce.
[7] A branch integrating the changes can be found at https://github.com/season-lab/fuzzolic.
[8] The artifact environment is available at https://github.com/ercoppa/debug-ce.

Table 2
Manually injected implementation gaps.

Injected bug	Engine	Implementation gap
WrongInstr	All	Subtraction is instrumented as an addition.
NoModel	SymCC	Skip analysis of function `memset`.
	SymQEMU Fuzzolic	Ignore effects of system call `lseek`.
WrongExpr	All	Subtraction expression is built as an addition expression.
AltWrongExpr	All	Sign-extend expression is built as a zero-extend expression.
WrongModel	SymCC	Incorrect modeling of the effects of `ntohl`.
	SymQEMU	Incorrect modeling of the effects of `lseek`.
	Fuzzolic	Incorrect modeling of the effects of `memcmp`.
WrongOpt[a]	SymQEMU	Seed random and $(B - A)$ rewritten as $(-A - B)$ instead of $(-A + B)$.
	Fuzzolic	Seed zero and $(A - B == 0)$ rewritten as $A == 0$ instead of $A == B$.
AltWrongOpt	SymQEMU	Seed zero and $(B - A)$ rewritten as $(-A - B)$ instead of $(-A + B)$.
	Fuzzolic	Seed random and $(A - B == 0)$ rewritten as $A == 0$ instead of $A == B$.
WrongPI	All	The branch is not negated when updating the path constraints.
WrongQuery	All	The branch is not negated when building the solver query.
WrongSmt	SymCC	Same as WrongExpr since it directly builds Z3 expressions.
	SymQEMU	Operator > is translated using `Z3_mk_bvslt` instead of `Z3_mk_bvsgt`.
	Fuzzolic	Another variant of S2A since it directly builds Z3 expressions.

[a] SymCC does not perform simplifications when using the simple backend.

After defining our setup (Section 5.1), we consider a simplified scenario (Section 5.2) that allows us to make a first validation of our strategies. In particular, in this scenario, we injected well-defined implementation gaps into the concolic frameworks and then we carry out concolic exploration over synthetic and simple programs. These experiments could simulate the introduction of silly errors during the development of a concolic framework, where we may expect users to test it with minimal programs.

We then consider a more complex scenario where developers run the concolic framework over real-world programs. We first report the number of inconsistencies detected by our strategies (Section 5.3) for the three concolic executors and then investigate a subset of these inconsistencies to identify their actual root cause.

5.1. Experimental setup

We executed our experiments within a Docker container based on Ubuntu 20.04 where we integrated the three concolic executors. The container was executed on a server equipped with two Intel Xeon Gold 6238R CPU @ 2.20 GHz and 640 GB of RAM. To make our experiments reproducible, we have released our experimental setup (Section 4), which also integrate the specific changes to the three concolic executors.

5.1.1. Simplified scenario

To perform a first validation of our strategies, we consider a simplified scenario where we artificially injected implementation gaps within the specific steps (Section 2.2) of a concolic executor. Table 2 reports a brief description of the manually injected bugs.

When considering a few specific steps, we opted into injecting slightly different implementation gaps to meet the specific nature and design traits of each concolic executor: e.g., for step S3, associated with the bug WrongModel in the table, SymCC does not model system calls and hence we aimed toward a user–space function such as `ntohl`, while SymQEMU only models a few system calls hence we aimed at breaking the modeling of `lseek`, finally, in the case of Fuzzolic, although this engine does not need to explicitly model user–space functions, it still prefers to model several user–space functions for efficiency reasons and we opted to break the modeling of `memcmp`.

To allow our strategies to possibly detect the inconsistencies during the concolic exploration, we devised one minimal program for each implementation gap. This program guarantees the activation of the relevant step affected by our changes during the exploration. For instance, for NoModel, we devised a minimal program using `memset` for SymCC and a minimal program using `lseek` for SymQEMU and Fuzzolic.

We avoided to consider large and complex programs as the targets of the exploration in this scenario because they could expose unknown implementation gaps already present in these concolic frameworks (as confirmed by Section 5.3), making it hard to understand whether a detected inconsistency was actually related to our injected implementation gaps.

5.1.2. Real-world scenario

When moving to a more realistic and challenging testing scenario, we focused on applications that were used in past experimental evaluations of the three concolic executors (Borzacchiello et al., 2021b; Poeplau and Francillon, 2020, 2021). In particular, we considered:

- `objdump` and `readelf` from GNU Binary Utils 2.34;
- `tcpdump` 4.9.3 with `libpcap` 1.9.1 statically linked;
- `bsdtar` from the Libarchive project commit `f3b1f9f239`;
- a PNG parser based on `libpng` 1.6.37;
- `tiff2pdf` from libTIFF 4.1.0.

As the input for the exploration, we considered syntactically minimal files often used in fuzzing experiments since they could be likely used by a developer when testing the concolic executor over the real-world program. Each application was analyzed over one seed with a 90-second timeout,[9] while for ChkPC and ChkInp checks, which may require additional executions, we allowed the container to run up to 8 h. The memory limit for each container was set to 8 GB.

5.2. RQ1: Can the strategies identify manually injected bugs?

Our first research question aims at validating whether our testing strategies can identify specific implementation gaps that we artificially injected in the concolic frameworks. Table 3 shows an overview of the effectiveness of our strategies when considering the bugs from Table 2. Several interesting insights can be derived when analyzing these results.

First, no single strategy is able by itself to identify all our injected bugs. Moreover, we may expect that any bug introduced at an early

[9] This is the default value for SymCC and SymQEMU when running in hybrid mode.

Table 3
Consistency strategies versus manually injects bugs.

Injected Bug	Engine	ChkExpr	FuzExpr	ChkPC	ChkInp	EvOpt	SmtOpt
WrongInstr	SymCC	✓	✓				
	SymQEMU	✓	✓	✓	✓		
	Fuzzolic	✓	✓		✓		
NoModel	SymCC	✓	✓		✓		
	SymQEMU	✓	✓		✓		
	Fuzzolic	✓	✓		✓		
WrongExpr	SymCC	✓	✓	✓	✓		
	SymQEMU	✓		✓		✓	✓
	Fuzzolic	✓	✓				
AltWrongExpr	SymCC		✓				
	SymQEMU		✓		✓		
	Fuzzolic		✓				
WrongModel	SymCC	✓	✓				
	SymQEMU	✓					
	Fuzzolic			✓			
WrongOpt	SymQEMU					✓	✓
	Fuzzolic			✓	✓	✓	✓
AltWrongOpt	SymQEMU						✓
	Fuzzolic				✓		✓
WrongPI	SymCC		✓	✓			
	SymQEMU			✓			
	Fuzzolic			✓			
WrongQuery	SymCC		✓		✓		
	SymQEMU		✓		✓		
	Fuzzolic				✓		
WrongSmt	SymCC	✓	✓	✓	✓		
	SymQEMU	✓	✓	✓			
	Fuzzolic	✓	✓		✓		

step of the concolic analysis would be detected by any check performed in later analysis steps. Unfortunately, this is not the case: for instance, when considering the bug NoModel, we can see that strategy ChkExpr is quite effective at detecting the inconsistency over the resulting wrong expression, however, when the wrong expression is added to the path constraints, it happens than ChkPC does not detect any problem. This suggests that the added expression, while partially wrong, does not make the path constraints immediately infeasible or inconsistent with the current path.

Second, strategy ChkPC has a great potential to find bugs. However, we need to remember that due to its design, it has some practical downsides: (a) it requires to query an SMT solver to generate new values for an expression, (b) each new value requires a new run of the concolic executor to check its consistency, (c) a tool cannot realistically enumerate all possible values for an expression due to time constraints, and (d) the solver can potentially give us values that are very similar to each other, possibly making hard to detect an inconsistency. For instance, for AltWrongExpr, ChkPC checks are crucial to detect such kind of bug, however, we had to generate up to 16 values for each expression to make the inconsistency appear in all three concolic executors. While one may be tempted to further increase such a threshold, this could soon lead to an unsustainable number of executions. We will provide a few concrete numbers when considering real-world programs.

Third, ChkInp is quite powerful but it cannot always detect a bug since its effectiveness depends on the number of branches for which the concolic executor can generate a satisfying assignment. Moreover, we observed in some cases that when the expressions are slightly wrong, the concolic executor may still be able to generate a valuable input able to visit the expected direction of a branch. Furthermore, we remark that each ChkInp check requires to perform an additional program run. However, differently from FuzExpr checks, as shown by the numbers in the next section, we believe ChkInp checks are still sustainable in most cases.

Fourth, ChkPC can miss several bugs. As for ChkInp, we observed that in several cases the bug was not making the path constraints infeasible or inconsistent. However, our programs were quite simple hence this strategy could show better results when considering programs with several nested branch conditions.

Fifth, EvOpt and SmtOpt were useful only in a few cases. However, they were spot on when considering WrongOpt and AltWrongOpt which were designed for validating exactly these two strategies. Notice that WrongExpr for SymQEMU is detected by EvOpt and SmtOpt because WrongExpr's bug affects one primitive of the expression builder that is also used by the expression optimizer.

5.3. RQ2: How consistent are the existing concolic engines?

We now consider a more realistic scenario where we look out for inconsistencies in the current implementations of the three concolic executors when running them over real-world programs. Table 4 reports the number of failed checks performed by each strategy when considering a specific concolic engine and a given program.

When focusing on strategy ChkExpr, SymCC and Fuzzolic can pass 99% of the checks, while SymQEMU fails 32% of them. For the first two concolic executors, this result is not surprising since they were evaluated in different previous works (Poeplau and Francillon, 2020; Borzacchiello et al., 2021b) using targets consistent with our program choice, likely fixing along the road several implementation bugs that may have emerged on these targets. However, there are still several inconsistencies left to investigate. The result on SymQEMU is instead more worrying as a large number of expressions seem to be inaccurate, possibly suggesting that SymQEMU is often wasting time during its analysis. Another interesting insight that emerges when observing the results related to `tiff2pdf` is that Fuzzolic is building way more expressions than the two other engines. We further elaborate on this experimental observation in Section 5.5.

When moving to the results from FuzExpr, we can see that this strategy is indeed quite powerful as it can reveal inconsistencies that

are missed by ChkExpr. For instance, both SymCC and Fuzzolic are able to pass 100% of ChkExpr checks on `readelf` but then fail to pass several checks generated by this strategy. Overall, the percentage of failed checks is 32%, 82%, and 11%, for SymCC, SymQEMU, and Fuzzolic, respectively. However, this (likely) improved effectiveness in detecting inconsistencies comes with a notable cost in terms of running time: for instance, SymCC performed 155,472 executions during ChkPC compared to six executions during ChkExpr. While later on in this section we report additional performance figures on the overhead from our testing strategies, we can easily claim that ChkPC requires *hours* compared to (a few) *minutes* of the other strategies. Moreover, we remark that the total number of executions depends on the number of alternative values generated with the solver over an expression. As for experiments from Section 5.2, we limited the number of values for the same expression to 16. We believe that this threshold should be hand-tuned by a developer depending on the time budget available for testing. Furthermore, we remark that, consistently with the practice adopted by SymCC and SymQEMU in hybrid mode, we aborted the concolic exploration over the initial seed, i.e., the one generating the alternative values for the expressions, after 90 seconds. Hence, SymQEMU performed fewer ChkPC checks than ChkExpr checks merely because during the 90 seconds it had to perform way more expensive queries than when using ChkExpr (which only requires to evaluate an expression).

The results on ChkPC and ChkInp are quite interesting as they clearly show that the path constraints are wrong in a significant number of cases regardless of the concolic engine. This inaccuracy leads the engines into generating inputs that are not able to bring the program to the expected branch directions. For instance, although SymCC is able to pass the majority of ChkPC checks, its generated inputs are often unable to reach the predicted program point. Moreover, when investigating the results for ChkPC, we noticed that some engines have debug capabilities to check at running time whether the path constraints are satisfiable: as pointed out in Section 3.3, besides being more expensive than ChkPC, this approach is also incorrect, possibly failing to detect bugs. For instance, SymCC can pass 378 ChkPC checks out of 382 on `objdump`. However, when testing a ChkPC variant based on the satisfiability of the path constraints, we observed that the passed checks were 382, i.e., 100%. Hence, SymCC is building path constraints that can be satisfied but are not representative (consistent) with the current path: e.g., in one case, we observed that it was asserting that the 40-th byte must be equal to 48, however, while this could be true for other paths, this was inconsistent with the seed used in our experiment.

Finally, results from EvOpt and SmtOpt do not indicate the presence of any inconsistency related to expression optimizations. While this was expected from SymCC since it relies only on the simplifying primitive from the SMT solver (which is unlikely to contain bugs), it was less expected for the two other concolic engines. While SymQEMU and Fuzzolic mostly devise simple rewriting rules, they nonetheless cope with a large number of simplification patterns (more than 40 in both frameworks), encoded through a significant number of lines of code (∼800 LOC in SymQEMU, ∼1500 LOC in Fuzzolic). Hence, we believe that these optimizations were carefully designed or, at least, reasonably tested. ChkInp checks were able to finish within our 90-second timeout for most targets and engines. Notable exceptions are for SymQEMU on `objdump`, `bsdtar`, and `libpng`. For instance, on `objdump`, ChkInp could only validate 7% of the optimizations before running out of time, demonstrating the benefit of having ChkPC checks, which although weaker can still provide valuable validation feedback even when ChkInp cannot complete (see Table 4).

Overhead. Table 5 provides an overview of the running time overhead resulting from our strategies during our experiments. In particular, for strategies ChkExpr, ChkPC, EvOpt, and SmtOpt, we report the slowdown,[10] while running the concolic engine with each technique enabled with respect to one where no technique is enabled. These experiments took at most 90 s as explained in Section 5.1.2. Differently, for FuzExpr and ChkInp, we report the total running time for the experiment, which measures the time spent for the initial run needed for generating the alternative inputs and the time spent for the additional executions. While our implementations are likely suboptimal and could benefit from optimizations, we believe that ChkExpr, ChkPC, and EvOpt are *reasonably* sustainable,[11] for a developer interested in consistently testing a framework. EvOpt and ChkPC appear to induce significantly higher overheads, likely suggesting that they should be enabled at specific times (e.g., after devising new optimizations). Finally, as easily predictable, FuzExpr requires hours when used by an engine due to the large number of executions spawned during the experiment. Hence, this strategy should be likely used in a limited number of tests, e.g., before the release of a new version of an engine.

5.4. RQ3: Can the identified inconsistencies point to actual bugs?

The results reported in the previous section are only valuable for a developer if they are pointing the finger toward actual implementation gaps of the three concolic engines. Since validating the correctness of the detected inconsistencies from all strategies would take (likely) years, we decided to focus on inconsistencies reported by strategy ChkExpr. To validate an inconsistency, we had to understand its root cause and devise appropriate fixes to demonstrate that we actually identified the correct problem. In particular, we repeated in a loop the following methodology: we considered the first inconsistency reported by ChkExpr during an experiment, devised a fix for it, and then repeated the experiment to obtain the up-to-date set of inconsistencies. Table 6 reports the number of failed checks from strategy ChkExpr before and after our full set of fixes. Notice that, after fixing a bug, one expression may be propagated in a different way, hence the total number of checks may increase, or even decrease (when the bug was making the engine propagate a symbolic expression in place of a concrete value). Overall, we did not hit any false positives (not even caused by non determinism). Moreover, although we do not provide objective measures, we can informally report that the fine granularity of our checks significantly helped to quickly identify the location of the root cause, as the failure happened quite close to the code introducing the inconsistency. We now briefly review our discoveries.

5.4.1. Inconsistencies in SymCC

Input generation in the presence of seek operations. The *simple* backend of SymCC does not correctly generate fresh symbolic bytes when a *seek* operation, such as `lseek`, is performed on the input file. Indeed, when a program *jumps* to a file offset X and reads one or more bytes from that position, SymCC implicitly concretizes, i.e., sets to `nullptr`, any input byte that was not explicitly read before the *seek* operation. We reported the issue and proposed a fix (Coppa, 2023e).

Conversion from bool to bitvector expression. SymCC was not correctly performing a conversion from a *boolean* expression to a *bitvector* expression. Indeed, the implementation was generating an *If-Then-Else* expression of the form *ITE(bool_expr,* `0x0`*,* `0x1`*)* where the resulting value (`0x0` or `0x1`) was always at least 8 bits long. This is incorrect since then conversion should generate a similar ITE expression but with

[10] Since ChkExpr ChkPC, EvOpt, and SmtOpt do not depend on the solving of the branch queries, we disabled them. Hence, our slowdowns are *pessimistic* as we may expect developers to possibly run the checks while also performing branch queries.

[11] ChkPC appears to be quite heavy in SymQEMU because the baseline is not calling the function `Solver::syncConstraints` which, however, would be called in a full concolic execution. Hence, the reported slowdown is quite *worst case*.

Table 4
Failed consistency checks when testing the three concolic executors.

Program	Engine	ChkExpr	FuzExpr	ChkPC	ChkInp	EvOpt	SmtOpt
objdump	SymCC	4/6029	22668/47621	4/382	20/57	0/419	0/419
	SymQEMU	60254/126781	75455/80954	997/1833	40/149	0/2042832	0/156645
	Fuzzolic	0/11594	1949/27052	0/1879	24/140	0/2209	0/2209
readelf	SymCC	0/4671	3449/52103	0/1101	59/160	0/1106	0/1106
	SymQEMU	222/12830	1544/6454	83/1262	32/72	0/24948	0/24948
	Fuzzolic	0/4147	242/12061	0/1016	5/124	0/1147	0/1147
tcpdump	SymCC	0/4830	8196/18514	0/446	18/106	0/452	0/452
	SymQEMU	869/3114	1567/4008	428/595	33/63	0/13165	0/13165
	Fuzzolic	178/7839	873/10060	291/563	1/53	0/2737	0/2737
bsdtar	SymCC	0/12787	44/15638	1/504	2/60	0/520	0/520
	SymQEMU	2/2017	3912/6902	0/358	87/114	0/1831	0/1741
	Fuzzolic	310/2784	2583/8111	816/976	80/102	0/2624	0/2624
libpng	SymCC	0/7360	17798/20892	0/705	82/94	0/744	0/744
	SymQEMU	2550/51697	116/1939	100/2211	55/82	0/64608	0/11383
	Fuzzolic	0/22893	2630/13846	0/8174	18/54	0/11121	0/11121
tiff2pdf	SymCC	13/308	60/704	2/39	2/8	0/42	0/42
	SymQEMU	0/397	16/192	0/173	1/6	0/409	0/409
	Fuzzolic	0/21884	1530/22592	0/11882	47/233	0/8337	0/8337

Table 5
Slowdown and overall running time of the different techniques.

Program	Engine	ChkExpr	FuzExpr	ChkPC	ChkInp	EvOpt	SmtOpt
objdump	SymCC	7.2×	4187 secs	2.5×	4.5 secs	28.3×	42.8×
	SymQEMU	1.9×	28800 secs	18.5×	1004 secs	29.8×	T/O
	Fuzzolic	1.1×	28002 secs	1.2×	452 secs	2.7×	33.7×
readelf	SymCC	3.3×	5144 secs	2.5×	23 secs	33.5×	62.0×
	SymQEMU	2.5×	894 secs	T/O	190 secs	10.8×	105.7×
	Fuzzolic	1.1×	9983 secs	1.1×	158 secs	11.5×	10.5×
tcpdump	SymCC	2.7×	3564 secs	1.6×	25.6 secs	14.6×	30.4×
	SymQEMU	1.7×	646 secs	48.1×	37 secs	54.2×	226.5×
	Fuzzolic	1.1×	24702 secs	1.0×	260 secs	4.2×	30.1×
bsdtar	SymCC	6.3×	1182 secs	1.5×	183 secs	2.7×	20.7×
	SymQEMU	1.4×	814 secs	4.6×	17.5 secs	20.7×	T/O
	Fuzzolic	1.6×	14246 secs	1.2×	336 secs	7.5×	26.9×
libpng	SymCC	3.3×	1665 secs	2.5×	8.7 secs	20.0×	47.9×
	SymQEMU	31.7×	293 secs	T/O	191 secs	90.9×	T/O
	Fuzzolic	1.1×	14130 secs	17.5×	59 secs	9.2×	T/O
tiff2pdf	SymCC	1.5×	27 secs	1.3×	0.5 secs	16.8×	155.3×
	SymQEMU	1.1×	23 secs	51.9×	5.5 secs	12.9×	46.6×
	Fuzzolic	1.1×	26396 secs	8.6×	267 secs	6.1×	T/O
Mean		2.2×	9149 secs	3.8×	179 secs	13.7×	45.9×

a resulting expression of size equal to 1 bit and then perform sign- or zero-extension based on the code context. The current implementation thus was not correctly supporting sign extension after a conversion: an 8-bit sign extension of a `true` value, after conversion, would be equal to the 8-bit `0x1`, while a correct implementation should generate `0xFFFF`. We reported the issue and proposed a fix (Coppa, 2022a).

Accessing bool expressions in memory. SymCC was not storing (reading) correctly boolean expressions to (from) its symbolic memory. Indeed, it was not performing a conversion from a boolean to a bitvector. This bug is complementary with the previous one and it could lead to a crash in some cases. We reported the issue and proposed a fix (Coppa, 2022c).

Instrumentation of conditional movement. SymCC was not propagating the symbolic expression in the presence of a conditional movement, i.e., `select` instruction in LLVM, which may be generated by, e.g., the C ternary operator. We reported the issue and proposed a fix (Coppa, 2022b).

Variadic functions. SymCC does not propagate the symbolic arguments of a variadic function. Unfortunately, the details behind variadic functions are platform specific. We reported the issue (Coppa, 2022d) and implemented a partial fix for Linux `x86_64` in our fork that can at least remove inconsistencies.

Side-effects of sprintf. SymCC does not provide a *model* for `sprintf`, ignoring its effects on the symbolic state. While the lack of a model is understandable due to the complexity of `sprintf`, it could make sense to have at least a concretization over the bytes written by `sprintf` to avoid inconsistencies. We reported the issue and proposed a fix (Coppa, 2023f).

Effects from uninstrumented library code. SymCC can only track the effects of code that has been instrumented at compilation time. Unfortunately, recompiling *all* system libraries for an application is impractical (Coppa et al., 2022). For instance, on program `libpng`, we could not fix all ChkExpr inconsistencies, as shown by Table 6, because several of them were due to uninstrumented code. While this is not a flaw that can be fixed, our strategies can at least help a developer understand when these ignored effects may have a negative impact.

5.4.2. Inconsistencies in SymQEMU

Program counter tracking. SymQEMU internally tracks which basic blocks are executed. This is useful when updating a coverage bitmap that is pivotal for avoiding redundant queries across different runs. Moreover, it is valuable when performing debugging (e.g., during our validation). Unfortunately, SymQEMU incorrectly derives the program

Table 6
Number of failed checks from strategy ChkExpr before and after our fixes. For SymCC on libpng, the number of inconsistencies increases because, after fixing some implementations gaps that were limiting the propagation of symbolic expressions, our approach was able to detect new inconsistencies resulting from the lack of instrumentation of library code (see discussion in *Effects from uninstrumented library code*).

Program	SymCC		SymQEMU		Fuzzolic	
	Before	After	Before	After	Before	After
objdump	4/6029	0/12812	60254/126781	0/15360	0/11594	0/11594
readelf	0/4671	0/6101	222/12830	0/23962	0/4147	0/4147
tcpdump	0/4830	0/4848	869/3114	0/24985	178/7839	0/8895
bsdtar	0/12787	0/12790	2/2017	0/2253	310/2784	0/4464
libpng	0/7360	752/9032	2550/51697	0/52838	0/22893	0/22893
tiff2pdf	13/308	0/295	0/397	0/410	0/21884	0/21884

counter. The issue has been already reported in the past (Haochen, 2023) and we proposed a fix (Coppa, 2023h).

Symbolic reasoning over QEMU helpers. SymQEMU ignores the effects of most QEMU helpers: these are used internally by QEMU to replicate the effects of architecture-specific instructions. A large number of helpers is used on targets, such as `i386` and `x86_64`, where they model, e.g., the division operation, vectorized instructions, and floating-point operations. As a mitigation, SymQEMU conservatively concretizes the QEMU temporary register emitted in output by a helper (if any). Unfortunately, this is not enough to avoid inconsistencies since several helpers have side effects directly over the memory. Moreover, concretizations may harm the overall effectiveness of SymQEMU. We have proposed to exploit SymCC for the instrumentation of these helpers. Our proposal (Coppa, 2023j) requires a few important changes in SymQEMU and we are waiting for the feedback from the developers before proposing a fix (however, our fork contains a first Proof-Of-Concept).

High part of an unsigned multiply operation. SymQEMU correctly builds a symbolic expression in the presence of the QEMU helper in charge of computing the high part (most significant bits) of a 128-bit unsigned multiply operation. However, since SymQEMU by default ignores the effects of most helpers, it concretizes the resulting output expression even in the case of an unsigned multiply operation. We reported the issue (Coppa, 2023i) and proposed a fix within our fork.

CLZ and CTZ operations. SymQEMU ignores CTZ and CLZ operations, possibly generating inconsistencies. We have reported the problem (Coppa, 2023k) and proposed a fix that removes the inconsistencies.

Sign extension when loading values from memory. SymQEMU does not correctly perform sign extension in some cases when the program is loading symbolic values from the memory. This issue was already reported in the past (Coppa, 2023g) and we proposed a fix (Coppa, 2023d).

Instrumentation of setcond instructions. The symbolic instrumentation of the `setcond` instruction can be incorrect in some programs when one QEMU temporary taken in input by the instruction is also used as the output QEMU temporary. We have reported the problem and proposed a fix (Coppa, 2023l).

5.4.3. Inconsistencies in Fuzzolic

Incorrect access to XMM registers in packuswb instruction. The symbolic model of the `i386` and `x86_64` instruction `packuswb` was incorrectly performing pointer arithmetic to reach the XMM registers. We proposed a fix (Coppa, 2023b), which will be part of the next release.

Incorrect instrumentation of punpck and packuswb instructions. The symbolic instrumentation of the `i386` and `x86_64` instructions `punpck` and `packuswb` was passing the wrong arguments to the model. We proposed a fix (Coppa, 2023c), which will be part of the next release.

Additional inconsistencies. The strategies described in this article were initially conceived during the development of Fuzzolic, helping its authors to detect a large number of bugs. Since we did not keep track in detail of their impact during the development, we do not discuss in detail these inconsistencies. However, as authors of Fuzzolic, we can at least confirm that these strategies can be extremely helpful while testing a concolic engine.

5.5. Discussion

The results presented in this section show that the consistency strategies described in this article can be valuable in finding implementation gaps in state-of-the-art concolic executors. While using these techniques during our experiments, we identified two notable downsides.

First, optimized real-world code may access uninitialized memory values. For instance, this can happen in the presence of vectorized instructions that may access more bytes than what is actually needed, exploiting alignment rules that make these additional bytes always fall within valid pages. This problem is also met when using memory error detectors such as Valgrind (Nethercote and Seward, 2007). Tweaks are needed to ignore inconsistencies of this kind.

Second, our consistency strategies do not help identify the lack of generation or propagation of symbolic expressions. In particular, we cannot detect when a concolic engine is not propagating an expression. This problem was evident when comparing the results on `tiff2pdf` from different concolic engines: Fuzzolic was clearly doing more work than SymCC and SymQEMU. After an investigation, we discovered that these two concolic executors are not handling the scenario where a file is mapped into a memory region, thus losing track of most data flows involving input bytes. We plan to submit fixes for this issue, however, we must observe that to notice such problem we had to compare the work of different tools, passing thus it unnoticed when using our strategies with a single concolic engine.

6. Conclusions

In this article, we have proposed several testing strategies for identifying implementation gaps in concolic engines. The main idea is to exploit the concrete state kept by an engine to identify inconsistencies in the symbolic state. By looking for inconsistencies at each operation of the analyzed program, we can catch the problem as soon as it affects the concolic execution and, hopefully, near the root cause of the bug.

We integrated our consistency strategies into three state-of-the-art concolic executors and showed that they reveal several inconsistencies when analyzing real-world programs. To demonstrate that these inconsistencies can point the finger toward actual implementation bugs, we analyzed a subset of them, identifying the root cause of the problem and proposing fixes to the original projects.

As future work, we believe that our strategies could still benefit from optimizations to reduce their overhead and possibly increase their practicality, favoring their adoption inside concolic engines. Strategy FuzExpr could benefit from a heuristic able to make the solver generate diverse concrete values for an expression. Moreover, our strategies cannot detect when an engine is failing to propagate a symbolic expression.

We believe that this problem could be tackled by envisioning strategies that exploit more lightweight data flow analysis, such as taint analysis, where mature implementations are already available to the research community.

CRediT authorship contribution statement

Emilio Coppa: Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization. **Alessio Izzillo:** Writing – review & editing, Writing – original draft, Validation, Methodology, Investigation, Conceptualization.

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.

Data availability

Code has been released on GitHub. URLs are included in the article.

Acknowledgments

This work was partially supported by Project PRIN 2022 FARE (202225BZJC, CUP B53D23012730006) and Project PRIN 2022 PNRR SETA (P202233M9Z, CUP B53D23026000001). Both projects are under the Italian NRRP MUR program funded by the EU - Next Generation EU.

References

Angelini, M., Blasilli, G., Borzacchiello, L., Coppa, E., D'Elia, D.C., Lenti, S., Nicchi, S., Santucci, G., 2019. SymNav: Visually assisting symbolic execution. In: Proc. of the 16th IEEE Symposium on Visualization for Cyber Security. VizSec '19, http://dx.doi.org/10.1109/VizSec48167.2019.9161524.

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

Bellard, F., 2005. QEMU, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. ATEC '05, URL: http://dl.acm.org/citation.cfm?id=1247360.1247401.

Borzacchiello, L., Coppa, E., D'Elia, D.C., Demetrescu, C., 2019. Reconstructing C2 servers for remote access trojans with symbolic execution. In: Cyber Security Cryptography and Machine Learning. CSCML '19, Springer International Publishing, http://dx.doi.org/10.1007/978-3-030-20951-3_12.

Borzacchiello, L., Coppa, E., Demetrescu, C., 2021a. Fuzzing symbolic expressions. In: Proceedings of the 43rd International Conference on Software Engineering. ICSE '21, http://dx.doi.org/10.1109/ICSE43902.2021.00071.

Borzacchiello, L., Coppa, E., Demetrescu, C., 2021b. FUZZOLIC: mixing fuzzing and concolic execution. Comput. Secur. http://dx.doi.org/10.1016/j.cose.2021.102368.

Borzacchiello, L., Coppa, E., Demetrescu, C., 2022. SENinja: A symbolic execution plugin for Binary Ninja. SoftwareX 20, http://dx.doi.org/10.1016/j.softx.2022.101219.

Cadar, C., Donaldson, A.F., 2016. Analysing the program analyser. In: Proceedings of the 38th International Conference on Software Engineering Companion. ICSE '16, Association for Computing Machinery, New York, NY, USA, pp. 765–768. http://dx.doi.org/10.1145/2889160.2889206.

Cadar, C., Dunbar, D., Engler, D., 2008a. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. OSDI '08, pp. 209–224, URL: http://dl.acm.org/citation.cfm?id=1855741.1855756.

Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R., 2008b. EXE: Automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. http://dx.doi.org/10.1145/1455518.1455522.

Cadar, C., Sen, K., 2013a. Symbolic execution for software testing: Three decades later. Commun. ACM 56 (2), 82–90. http://dx.doi.org/10.1145/2408776.2408795.

Cadar, C., Sen, K., 2013b. Symbolic execution for software testing: Three decades later. Commun. ACM 56 (2), 82–90. http://dx.doi.org/10.1145/2408776.2408795.

Chen, J., Han, W., Yin, M., Zeng, H., Song, C., Lee, B., Yin, H., Shin, I., 2022. SYMSAN: Time and space efficient concolic execution via dynamic data-flow analysis. In: 31st USENIX Security Symposium. USENIX Security 22, URL: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-ju.

Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L., 2020. A survey of compiler testing. ACM Comput. Surv. 53 (1), http://dx.doi.org/10.1145/3363562.

Coppa, E., 2022a. Issue #108 in SymCC. https://github.com/eurecom-s3/symcc/issues/108.

Coppa, E., 2022b. Issue #109 in SymCC. https://github.com/eurecom-s3/symcc/issues/109.

Coppa, E., 2022c. Issue #112 in SymCC. https://github.com/eurecom-s3/symcc/issues/112.

Coppa, E., 2022d. Issue #120 in SymCC. https://github.com/eurecom-s3/symcc/issues/120.

Coppa, E., 2023a. Artifact for this article. URL: https://github.com/ercoppa/debug-ce.

Coppa, E., 2023b. Bugfix in Fuzzolic: accesses to XMM registers. https://github.com/season-lab/qemu/commit/00b64ca10.

Coppa, E., 2023c. Bugfix in Fuzzolic: instrumentation of punpck and packuswb instructions. https://github.com/season-lab/qemu/commit/5a9021f0f41.

Coppa, E., 2023d. Fix for Issue #14 from SymQEMU. https://github.com/eurecom-s3/symqemu/issues/14#issuecomment-1499087161.

Coppa, E., 2023e. Issue #136 in SymCC. https://github.com/eurecom-s3/symcc/issues/136.

Coppa, E., 2023f. Issue #137 in SymCC. https://github.com/eurecom-s3/symcc/issues/137.

Coppa, E., 2023g. Issue #14 in SymQEMU. https://github.com/eurecom-s3/symqemu/issues/14.

Coppa, E., 2023h. Issue #21 in SymQEMU. https://github.com/eurecom-s3/symqemu/issues/21.

Coppa, E., 2023i. Issue #23 in SymQEMU. https://github.com/eurecom-s3/symqemu/issues/23.

Coppa, E., 2023j. Issue #24 in symqemu. https://github.com/eurecom-s3/symqemu/issues/24.

Coppa, E., 2023k. Issue #25 in SymQEMU. https://github.com/eurecom-s3/symqemu/issues/25.

Coppa, E., 2023l. Issue #26 in SymQEMU. https://github.com/eurecom-s3/symqemu/issues/26.

Coppa, E., Yin, H., Demetrescu, C., 2022. SymFusion: Hybrid instrumentation for concolic execution. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ASE '22, http://dx.doi.org/10.1145/3551349.3556928.

Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., Yang, X., 2012. Testing static analyzers with randomly generated programs. In: Proceedings of the 4th International Conference on NASA Formal Methods. NFM '12, Berlin, Heidelberg, http://dx.doi.org/10.1007/978-3-642-28891-3_12.

Daniel, B., Dig, D., Garcia, K., Marinov, D., 2007. Automated testing of refactoring engines. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. In: ESEC-FSE '07, http://dx.doi.org/10.1145/1287624.1287651.

De Moura, L., Bjørner, N., 2011. Satisfiability modulo theories: Introduction and applications. Commun. ACM 54 (9), 69–77. http://dx.doi.org/10.1145/1995376.1995394, URL: http://doi.acm.org/10.1145/1995376.1995394.

Godefroid, P., Klarlund, N., Sen, K., 2005. DART: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '05, http://dx.doi.org/10.1145/1065010.1065036.

Godefroid, P., Levin, M.Y., Molnar, D.A., 2008. Automated whitebox fuzz testing. In: Proc. Network and Distributed System Security Symp.. NDSS '08, URL: http://www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf.

Godefroid, P., Levin, M.Y., Molnar, D., 2012. SAGE: Whitebox fuzzing for security testing. Commun. ACM http://dx.doi.org/10.1145/2093548.2093564.

Haochen, 2023. Issue #10 in SymQEMU. https://github.com/eurecom-s3/symqemu/issues/10.

Kapus, T., Cadar, C., 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. In: ASE 2017, http://dx.doi.org/10.5555/3155562.3155636.

Lattner, C., Adve, V., 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization. CGO 2004, pp. 75–86. http://dx.doi.org/10.1109/CGO.2004.1281665.

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A., 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1186–1189.

Myers, G.J., Sandler, C., Badgett, T., 2011. The Art of Software Testing. John Wiley & Sons.

Nethercote, N., Seward, J., 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '07, http://dx.doi.org/10.1145/1250734.1250746.

Pasareanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M., 2008. 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. http://dx.doi.org/10.1145/1390630.1390635.

Poeplau, S., Francillon, A., 2020. Symbolic execution with SymCC: Don't interpret, compile!. In: 29th USENIX Security Symposium. USENIX Security 20, URL: https://www.usenix.org/system/files/sec20-poeplau.pdf.

Poeplau, S., Francillon, A., 2021. SymQEMU: Compilation-based symbolic execution for binaries. In: Network and Distributed System Security Symposium.

Roy, C.K., Cordy, J.R., 2009. A mutation/injection-based automatic framework for evaluating code clone detection tools. In: 2009 International Conference on Software Testing, Verification, and Validation Workshops. pp. 157–166. http://dx.doi.org/10.1109/ICSTW.2009.18.

Sen, K., Marinov, D., Agha, G., 2005. CUTE: A concolic unit testing engine for c. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: ESEC/FSE-13, http://dx.doi.org/10.1145/1081706.1081750.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G., 2016. SoK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy. http://dx.doi.org/10.1109/SP.2016.17.

Wu, J., Hu, G., Tang, Y., Yang, J., 2013. Effective dynamic detection of alias analysis errors. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2013, http://dx.doi.org/10.1145/2491411.2491439.

Yang, X., Chen, Y., Eide, E., Regehr, J., 2011. Finding and understanding bugs in c compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11, http://dx.doi.org/10.1145/1993498.1993532.

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

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

Alessio Izzillo is Ph.D. student in Engineering in Computer Science at Sapienza University of Rome. He has been a participant of CyberChallenge.IT 2019. His research interests include cybersecurity, vulnerability detection, and firmware analysis.