SeaCoral: A Collaborative Test Generation Toolset for Industrial Orchestration of Testing Tools

Nicolas Berthier[1][0000−0002−0933−8193], Steven de Oliveira[1][0000−0003−1683−5902], Nikolai Kosmatov[2][0000−0003−1557−2813], and Delphine Longuet[2][0000−0002−8394−276X]

[1] OCamlPro, Paris, France
{nicolas.berthier,steven.de-oliveira}@ocamlpro.com
[2] Thales Research and Technology, cortAIx Labs, Palaiseau, France
{nikolai.kosmatov,delphine.longuet}@thalesgroup.com

Abstract. Formal methods have been successfully used to develop advanced test generation techniques. While various test generation tools and test coverage criteria have been proposed, their adoption in industrial practice remains fragmented. This paper presents SeaCoral, a novel open-source toolset for testing C programs, which aims to facilitate the industrial application of diverse testing technologies by integrating them within a single framework. SeaCoral offers a comprehensive set of testing services, ranging from the specification of test objectives for selected test coverage criteria, through test case generation and the detection of uncoverable objectives, to the measurement of the resulting coverage. It integrates a rich set of analyzers: the fuzzer libFuzzer, the model-checker CBMC, the dynamic symbolic execution tool KLEE, and the static analyzer LUncov for detecting uncoverable test objectives. These analyzers enrich each other's results through a shared store of test objectives and a shared corpus of test cases. Particular attention is paid to the careful handling of runtime errors and initialization functions, which are essential for industrial applications. Initial experiments confirm the benefits of the proposed toolset.

Keywords: test generation toolset, fuzzing, model checking, symbolic execution, static analysis, test coverage criteria, KLEE, libFuzzer, CBMC, LUncov, SeaCoral.

1 Introduction

Testing [4, 31, 34, 42] remains the most popular software validation technique in industry. Many efficient testing tools have been developed for C programs, based either on formal methods based approaches such as *symbolic execution*, *model checking* and *constraint solving*, or on more heuristic techniques like *fuzzing* (e.g. [1, 15–17, 19, 21, 37, 40]). Various *test coverage criteria* (a.k.a. *adequacy criteria* or *test criteria*) [2, 4, 18, 27, 42] have been proposed to define test objectives and to evaluate the level of code coverage of a given test suite. Achieving a coverage criterion like statement coverage (all statements are executed at least once) or decision coverage (the two branches of all conditional statements are executed at least once) is often used as a confidence metrics for stakeholders. Moreover, some of these criteria are required by existing standards [22, 25] for critical software.

Except for fuzzing, the use of these tools in industrial development processes remains limited. Our experience of industrial application suggests that this is due to several reasons: (i) the diversity of the tools and of their approaches makes them difficult to take in hand, each one having its own specifics; (ii) the more user-friendly tools offer light coverage guaranties while an important part of the setup still remains manual; (iii) the tools offering strong guaranties, based on formal methods, need manual writing of harnesses and drivers, as well as fine understanding of the underlying techniques to achieve efficient configuration; (iv) the results come in different tool-dependent formats that are not immediately usable in testing campaigns. The purpose of this work is *to facilitate industrial collaborative applications of existing testing technologies by integrating them within the same framework*. The objective is to propose a simplified access to these different tools through a common interface that minimizes the need for user configuration, and to achieve collaboration between them in such a way as to combine their forces and compensate their weaknesses.

Testing Tool Combination. Platforms simplifying access to several tools through a common interface with minimal configuration exist for fuzzing, for example, OSS-Fuzz [24] proposed by Google, and OneFuzz [33] proposed by Microsoft (which is now dedicated to internal use only). Generalizing this idea to other kinds of testing tools, DeepState [23] offers a unified framework for heterogeneous testing tools, including symbolic execution and fuzzing tools for C/C++. However, the primary objective of these testing platforms is vulnerability detection rather than coverage, and they do not provide mechanisms for enabling tools to collaborate on the same code.

Beyond platforms that allow multiple tools to be used side by side, numerous research efforts focus on combining testing techniques themselves, particularly to enhance fuzzing. It is well known that a major limitation of fuzzing lies in program conditions that have very few satisfying inputs, which are unlikely to be discovered through random mutation. Consequently, combining fuzzing with model checking or symbolic execution is a natural approach. This has led to several tools, such as VeriFuzz [32], FuSeBMC [3] or FDSE [41], which perform remarkably well in the annual testing competition TestComp [38]. Another approach, in which a symbolic execution engine is designed to run in parallel with an existing fuzzer, is exemplified by SymCC [35].

Our goal is not to develop yet another hybrid testing tool, but rather to provide a platform that leverages existing tools and enables them to collaborate on the same code base through shared test objectives.

Labels for Collaborative Test Generation. Recent work [7, 8] proposed a generic way to specify test objectives for various coverage criteria with so-called *labels*. A *label* ℓ is a pair (l, e) where l is a location in the source code and e a boolean expression. *Covering* a label ℓ corresponds to finding a test that reaches location

```
//l1: x!=0 && y!=0
//l2: !(x!=0 && y!=0)
if (x!=0 && y!=0)
  ...
```

```
//l1: x!=0 && y!=0
//l2: x!=0 && y==0
//l3: x==0 && y!=0
//l4: x==0 && y==0
if (x!=0 && y!=0)
  ...
```

Fig. 1: Labels for DC (Decision Coverage) on the left, and MCC (Multiple Condition Coverage) on the right.

l while evaluating expression e to true. Satisfiability of many coverage criteria can be reduced to the search of a set of tests covering a given set of labels (e.g. for decision coverage, condition coverage, weak mutations, some variants of the modified condition-decision coverage (MC/DC) [8]). Figure 1 illustrates labels for two usual coverage criteria: decision coverage (DC—on the left) is specified by two test objectives for each conditional statement, where the condition must be evaluated to true and false; multiple condition coverage (MCC—on the right) is specified by 2^n test objectives for each conditional statement composed of n elementary subconditions, since it requires to cover all combinations of truth values of these subconditions.

LTest [5, 8] introduced a first testing toolset, where labels were used to express test objectives for various test coverage criteria and to measure the coverage of a test suite, while a dynamic symbolic execution (DSE) tool PathCrawler [39] was used for test generation and static analyzers of the Frama-C platform [10, 26] were used to find uncoverable test objectives. It attracted industrial interest [9]. In particular, an experiment on a large industrial code [28] showed the high potential of running several different tools in sequence on a shared version of the code and test objectives, each tool building on the results obtained by the previous one. Tests were generated for 99% of labels for the MC/DC criterion on 82,000 lines of C code, while the remaining labels were proved uncoverable, thus showing the clear interest of such a tool combination for industrial applications.

However, this approach, based on LTest and PathCrawler, has shown important limitations for larger applications. First, PathCrawler remains closed-source and is not supported anymore today. Second, LTest did not integrate other tools, such as model checkers or fuzzers. To circumvent these limitations, a recent work [13] proposed an adaptation of KLEE [15], a popular open-source test generator, to efficiently support test objectives expressed by labels of LTest. It shows that test generation guided by labels reaches a better coverage (often with fewer tests and in limited time), and it demonstrates that other tools can be integrated into the LTest framework.

Contributions. This paper presents SeaCoral [36], a novel open-source toolset for testing C programs. It was designed and developed in a collaboration between Thales and OCamlPro, based on clearly identified industrial needs. The purpose of SeaCoral is to gather the best ideas of previous efforts and to facilitate industrial applications and collaborations of mature testing tools by integrating them into an all-in-one testing platform. SeaCoral offers a complete range of automated testing services, including specification of test objectives for a chosen test coverage criterion, test case generation, detection of *infeasible* (i.e. uncoverable) objectives, and measurement of code coverage of the resulting test suite. It integrates several analyzers based on existing tools: a fuzzer libFuzzer [37], a model checker CBMC [19], a dynamic symbolic execution tool KLEE [15], and a static analyzer for detecting infeasible test objectives LUncov [6, 30]. To the best of our knowledge, SeaCoral is *the first toolset offering such a large panel of testing services and efficient open-source tools with a common management of test objectives, test suites and coverage,* and easily extensible thanks to being fully open-source.

SeaCoral generates a suitable harness for each analyzer, and provides a generic mechanism for specifying test objectives and for coverage measurement (based on the notion of *labels* [7]). Moreover, the analyzers can collaborate either by sequential or parallel executions. They enrich each other's results by means of: (i) a shared store of test objectives with their statuses; and (ii) a corpus of tests that enables analyzers to exploit already generated tests. This generated corpus is then translated in a unified readable format (a C file), thus facilitating their reuse outside of the tool. A particular attention is paid to carefully handling runtime errors (RTEs), test initialization functions, and usual preconditions (e.g. variable ranges or array sizes), which are important ingredients for industrial applications. SeaCoral required a significant development effort: 25K lines of OCaml, and 24 person-months. The tool is available as a docker image to ease its use in continuous integration and development chains. The companion artifact [12] includes a docker image with the tool and the illustrative examples of this paper.

Initial experiments confirm the benefits of the toolset. The use of each individual analyzer is much easier through SeaCoral, since most of the manual preparation and configuration required is automated (harness generation, precompilation, output formatting). Moreover, the collaboration mechanisms of SeaCoral enable users to execute analyzers sequentially or even in parallel for some of them, so that they enrich each other's results and achieve high coverage.

2 Collaborative Test Generation in Industrial Context

We first present a selection of well-established test-generation tools that covers a large range of analysis techniques. We then advance several factors that impair their adoption in industrial contexts and their interoperability.

2.1 Available Test Generation Tools

CBMC [19] is a bounded model-checker for C and C++ programs. It performs a reachability analysis that can be used to statically check the validity of user-provided assertions. In a first mode, called *assert* mode, the tool tries to prove that every assertion of the program holds and exhibits a counter-example for each falsifiable assertion. In a second mode, called *cover* mode, CBMC automatically generates test cases that satisfy a given code coverage criterion among several usual coverage criteria such as locations, conditions, or decisions, but also user-defined test objectives. In cover mode, CBMC maximizes the coverage of the given criterion by generating several valuations for the inputs of the tested function.

KLEE [15] is a popular test generation tool for C programs based on dynamic symbolic execution. It supports a single coverage criterion, which is the all-path criterion, meaning that its aim is to cover all the feasible execution paths of the (LLVM bitcode of the) program under test. KLEE performs a symbolic execution of the program according to various strategies, and generates a test for each path found to be feasible.

libFuzzer [37] operates by randomly mutating a corpus of test inputs that are stored using arrays of bytes. The fuzzing process consists in a random exploration of the set of test inputs by means of mutations and cross-overs. Each input is associated with a score, that is typically derived from the set of branches and instructions that it executes in the analyzed program. This score influences the likelihood for an input to be subject to a mutation or cross-over. In addition, libFuzzer is able to monitor executed instructions and derive (from comparison instructions for instance) a set of constants that it then uses to perform mutations on the inputs.

LUncov is part of the LTest toolset [5, 8]. It efficiently detects uncoverable labels from the results of Frama-C analyzes [6, 30], and more precisely of its plugins Eva (based on abstract interpretation) and WP (based on deductive verification).

2.2 Factors that Impair Industrial Adoption

While the above analyzers have reached an industrial maturity level, their use in industrial processes remains limited. Main factors that hinder a wider adoption include the need of expert knowledge to fine-tune the configuration knobs for the analyzers and to write suitable test harnesses, and the lack of available tooling for effectively exploiting their results.

Test generation and code analysis tools rarely work directly on a raw codebase. Instead, they usually require a proper initialization of the inputs of the analyzed program in order to take into account its actual invariants and preconditions. For example, in C, a pointer that is given as an argument of a function can either be a raw pointer or an array of unknown length. It is common in the latter case to have a dedicated argument representing the length of the array.

The definition of the program inputs is done via a *harness*, that is, a specialized program that creates the inputs of interest (some of which may need to be dynamically allocated) and their constraints before calling the analyzed function. While some analyzers like CBMC provide their own harness generation utility, most other tools require tedious manual work for writing harnesses. This phase is even more critical for fuzzers, for which the harness typically needs to convert an array of bytes into a suitable input data structure for the analyzed program. In every case, writing a test harness for programs that manipulate complex data structures with pointers is challenging. Automatic generation of harnesses is still an ongoing research problem [29].

Tuning configuration knobs for an analyzer often requires expert knowledge on the underlying analysis technique. For instance, a bounded model-checker like CBMC is intrinsically parameterized by a limit to the number of loop unfoldings. KLEE's exploration of all execution paths of the program under test sometimes needs a change of strategy to be more efficient. Fuzzers typically offer configurable heuristics that can help them in exploring paths more efficiently.

Finally, none of the aforementioned analyzers returns any actual test in C that can be readily compiled and replayed. Even more, each of them has a quite different output format. In cover mode, CBMC outputs the list of values for the inputs that were marked of interest with a dedicated construct in the harness. By contrast, in assert mode, CBMC constructs the full program inputs via a sequence of assignments in a counter-example that describes corresponding execution path. KLEE outputs a set of `.ktest` files, an internal format that consists in a series of symbolic names (human-readable strings) and binary data. libFuzzer entirely relies on unstructured arrays of bytes. At last, LUncov produces a CSV file that lists locations at which labels are proved uncoverable.

2.3 Factors that Hinder Collaboration between Analyzers

Collaboration between the analyzers is first impaired by the lack of agreement on the definition of some coverage criteria: even the definition of a simple criterion like branches, understood at the level of LLVM bitcode, does not usually correspond to branches at the source code level (not to mention the case of lazy comparison operators in C).

Next, collaborating by re-using test suites requires common interchange formats for representing the tests, and a set of converters that translate tests to and from the format that each tool accepts or produces. The issue of test conversion is even more striking in the case of fuzzers (and, to a lesser extent, of KLEE), as the format of the tests they produce, or accept as seeds, heavily depends on the test harness.

As is, none of the aforementioned analyzers offer any means to exploit already proven facts about labels. The efficiency of an analyzer may nonetheless be improved if it is able to exploit the fact that some labels have already been covered by a test. Likewise, the uncoverablility of a label may inform analyzers about branches that need not be explored.

At last, analyzers must be cautiously configured to make sure they work under the same hypotheses. Each tool makes assumptions on the analyzed program and adopts default settings that may not be equivalent. For instance, a static analyzer may consider that a branch is unreachable, for example, because it is preceded by an undefined behavior like a buffer overflow (hence does not meet the C standard), while a fuzzer generates a test that actually executes that branch.

SeaCoral facilitates the industrial adoption of these tools by automating the generation of test harnesses and offering a limited set of configuration settings. It is easy to use, and generates tests directly as C files. SeaCoral relies on the notion of labels to define shared tests objectives for every analyzer it integrates.

It additionally enables effective collaboration between the analyzers via a shared store of label statuses, and a shared corpus of test cases.

3 SeaCoral on a First Example

To emphasize SeaCoral's ease of use, we illustrate its usage and results with the example of a flasher manager coming from a real-time industrial C program embedded in a car computer. Generated from a Matlab Simulink design [20], it consists of 650 lines of code, with a large loop of 450 lines that is executed at each clock cycle. The function under test checks at runtime that the warning command has priority over other flashing commands. It takes as a parameter a fixed-sized array of 10×6 boolean inputs, representing 6 input signals over a sliding window of 10 cycles (that is, 60 values). Along with 110 global variables, the program has $60 + 110 = 170$ test inputs.

The inputs have rather simple datatypes without complex invariants, so a fuzzer should be able to easily generate a set of tests that cover a fair portion of the program. However, some execution paths are intricate and need constraint solving to be covered. Therefore, we use SeaCoral with libFuzzer first, to quickly get a coverage of the easiest paths. Then we use SeaCoral with a formal tool, for example CBMC, to complete the coverage with tests for rare branches. To be sure that the uncovered test objectives are not coverable, we can even use CBMC in its *assert* mode to prove uncoverability.

SeaCoral works as a command-line tool that can be configured via command-line options or a configuration file. It should be given at least a C program, an entrypoint (the function under test), a coverage criterion, and a set of analyzers to execute on this problem. We launch the following command to execute SeaCoral with libFuzzer on this program:

```
> seacoral --files flashermanager.c --entrypoint prio_warning --criterion MCC
          --tools libfuzzer --libfuzzer-runs 100000
```

The option `--files` accepts a list of files, one of them containing the function under test given with `--entrypoint`. The coverage criterion given by `--criterion` can be any criterion expressible by labels in LTest [5, 8].[3] Here we choose the multiple condition coverage criterion (cf. Fig. 1). The option `--tools` accepts a list of analyzers among the available ones (CBMC, KLEE, libFuzzer and LUncov), which are by default run in parallel. With the optional parameter `--libfuzzer-runs`, we instruct libFuzzer to perform 100 000 runs.

Without any additional configuration, SeaCoral outputs the following summary of its execution after 3.7 seconds:

```
Coverage statistics for 'prio_warning':
  cov: 103 (88.8%) uncov: 0 (0.0%) unkwn: 13 (11.2%) with 9 tests
Covered labels: {1, 3, 5, 6, 7, 8, 9, ..., 111, 112, 113, 116}
Uncoverable labels: {}
Crash statistics: rte: none
```

[3] See `https://git.frama-c.com/pub/ltest/lannotate/-/blob/master/doc/criteria.markdown` for the list of all coverage criteria supported by LTest.

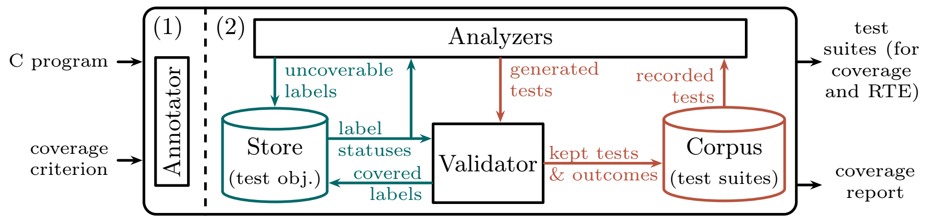

Fig. 2: Overview of SeaCoral with (1) Preparation phase and (2) Analysis phase.

The MCC criterion applied from this entrypoint produces 116 test objectives by adding labels to the whole program. Among these test objectives, 103 were reached by the 9 generated tests (status 'cov'), while 13 test objectives remain unreached (status 'unkwn'). No test objective was proved uncoverable (status 'uncov'), and no runtime error (RTE) was found.

To be able to complete this coverage, we launch SeaCoral a second time on the same file, entrypoint and criterion, this time with CBMC:

```
> seacoral --files flashermanager.c --entrypoint prio_warning --criterion MCC
          --tools cbmc --cbmc-mode assert --cbmc-unwind 61
```

We configure CBMC in *assert* mode, meaning that, at the same time, it tries to generate tests for coverable test objectives and to look for uncoverable ones. We set its *unwind* parameter to 61 $(10 \times 6 + 1)$ to make it unfold the loops of the program enough times for its bounded analysis to be complete (see Sect. 5 for details on the use of CBMC). This is one of the very few SeaCoral parameters requiring a bit of user knowledge about the analyzer since it cannot be automatically computed. With this configuration, CBMC resumes the previous execution of SeaCoral with the aim of completing the first coverage with additional tests or uncoverability proofs. In 1.9 seconds, it covers the 2 last coverable test objectives with 2 tests and proves the remaining 11 to be uncoverable.

In comparison, CBMC in *assert* mode launched alone on this program gives the same results, but in 20 seconds, instead of less than 6 seconds when the two analyzers are run in sequence. We explain below how the SeaCoral architecture enables such a smooth collaboration between its analyzers.

4 Design and Implementation of SeaCoral

Overview. SeaCoral operates on a given coverage criterion, a list of C files, and the name f of an *entrypoint* function to be tested. The main processing phases and components of SeaCoral are shown in Fig. 2.

At the *preparation phase* (1), the provided source code with its entrypoint and the coverage criterion are given to an *annotator*. First, it relies on LAnnotate module[4] of LTest [5, 8] to produce a version of the source code annotated with labels that encode the coverage criterion (cf. Fig. 1). The labels are then

[4] We extended this module to generate labels transitively in the callees of the entrypoint.

used to initialize a *store* of test objectives that associates each label with its coverage *status*. The preparation phase ends with the *generation of test harnesses* dedicated to *each analyzer*. For a symbolic analyzer, such a harness performs the symbolization of inputs (including global variables). For a fuzzer, the harness translates the array of bytes that encodes the inputs into values for these inputs. Besides, each harness instruments labels to make the analyzer target only the yet-undecided test objectives.

During the *analysis phase* (2), analyzers interact with the store, a *corpus of recorded tests*, and an additional *validator* component. The corpus gathers the set of recorded tests T, and associates each test with an *outcome* that belongs to two categories, thereby partitioning the set T into two disjoint subsets:

T_{cov}: the set of *label-covering* tests that terminate successfully;

T_{err}: the set of *RTE-triggering* tests inducing undefined behaviors or errors.

To avoid reporting an RTE multiple times, we identify each RTE with an *error summary*, that comprises a broad RTE categorization tag (arithmetic error, memory access error, etc), and a location in the code where the error occurs.

The validator plays a central role in ensuring the following key invariants:

($\mathfrak{I}_1$) labels reported as covered in the store are actually covered by at least one test in T_{cov};

($\mathfrak{I}_2$) any reported RTE is attached to a statement *in the original source code*;

($\mathfrak{I}_3$) any reported RTE is triggered by exactly one test in T_{err}.

Before making its way into the corpus, any test generated by an analyzer is given to the validator, which is the only component that may add a test into the corpus and record a label as covered in the store. This ensures ($\mathfrak{I}_1$). Analyzers are however able to directly report labels as uncoverable.

The Validator. Analyzers may generate a test simply because it reaches a label, without assessing whether it terminates successfully or identifying an error (if any). Several tests can trigger the *same* RTE, i.e. the same error at the same location in the source code. Industrial applications of testing tools need precise handling of RTEs. The validator provides a unified replay mechanism for (i) *coverage validation*, and (ii) if an RTE occurs, a precise *RTE identification*.

Coverage validation collects the labels that are covered by a given test t using an *instrumented validation program*, that: (i) evaluates the tested function on t; and (ii) performs runtime detection of RTEs using LLVM sanitizers. If it terminates successfully, and if t covers additional labels, then t is inserted into the corpus and the statuses for newly covered labels are submitted into the store. If the program terminates successfully and t covers no new labels, it is discarded.

An undefined behavior may be induced by the evaluation of a label condition that does not correspond to an RTE in the original source code. Consider for instance a condition `(y != 0 && x/y != 0)`. The annotated code may contain a label with condition `(x/y != 0)`, which triggers a division by 0 when `y == 0`, while it cannot occur in the original code due a lazy evaluation of `&&`.

To filter out such spurious RTEs, any test t for which coverage validation errors out, is subject to a second execution with an *RTE-identification program* where label conditions are not evaluated. If it terminates successfully, then t is discarded. Otherwise, the RTE is legitimate and $(\mathfrak{I}_2)$ is satisfied: the validator parses the associated sanitization report to extract a precise error summary. To ensure $(\mathfrak{I}_3)$, t is inserted into T_{err} if this summary has not yet been recorded.

Collaboration Mechanisms. Both the store and the corpus are used across (consecutive or parallel) launches of analyzers on the same program, entrypoint, and coverage criterion. In addition, SeaCoral enables any analyzer to reuse recorded tests. It achieves this via dedicated converters that perform the translations of tests to and from each analyzer's input and output formats.

5 Integration of Analyzers

Each SeaCoral analyzer is integrated by means of a dedicated *integration module*. Such a module automates the generation of test harnesses, that define the inputs of interest of the function under test (function arguments and global variables) and their constraints, and that translates the labels into targets for the analyzer. An integration module for an analyzer additionally performs all the necessary preparation phases like compilation or instrumentation of sources, and runs the analyzer itself with the relevant options so that it uses the results of previous runs and contributes only with valid tests for yet uncovered test objectives. At last, such a module handles the analyzer's outputs to consolidate all results in a unified format. We describe in this section some technical details of integration of each of the four analyzers of SeaCoral. The reader is not required to fully understand these details to use the tool and to follow the rest of the paper.

CBMC Integration. SeaCoral is able to interact with CBMC [19] using two different modes, *cover* and *assert* (selected with option `--cbmc-mode` on the command-line). The *cover* mode relies on the test generation capabilities of CBMC while the *assert* mode relies on its reachability analysis.

In *cover* mode, CBMC automatically generates test cases that satisfy a given code coverage criterion. It supports several usual coverage criteria but also user-defined test objectives defined by a built-in `__CPROVER_cover(cond)` statement. Obviously, these objectives have a direct correspondence with labels in SeaCoral. However, to ensure CBMC generates tests that terminate successfully, the harnesses generated by SeaCoral for CBMC in this mode associate each label with a Boolean flag that is only assigned whenever the label is reached and its condition holds. `__CPROVER_cover` calls are then placed at the end of the harness, guarded with its respective flag. We illustrate the treatment of labels using the `cov_label` macro for CBMC cover mode in Listing 1.1.

In *assert* mode, CBMC performs a reachability analysis that can be used to statically check the validity of user-provided assertions defined by a built-in `__CPROVER_assert(cond)` statement. This mode treats each assertion independently, unlike the cover mode that tries to solve all the test objectives in the same analysis. The assert mode is used in SeaCoral both to generate tests and to detect uncoverable test objectives, by translating each label into an assertion for

Listing 1.1: Label implementation used in harnesses for CBMC.

```
/* One Boolean variable (flag) per label */
static char __sc_cbmc_covered_1 = '\000'          // Flag for label 1
static char __sc_cbmc_covered_2 = '\000'          // Flag for label 2

/* We update the label's flag in the macro */
# define cov_label(expr, id, ...)                \
  do {                                            \ /* Set the flag for label id */
    if (expr) __sc_cbmc_covered_##id = '\001';  \ /* if expr is satified       */
  } while (0)                                       /* at the current location   */

/* Function to test, annotated with cov_label(expr,id,...) for each label */
void annotated_entrypoint() {
  ...
}

int main () {
  /* Initialization of variables (not detailed here) */
  ...
  annotated_entrypoint();
  /* Checking if a label is covered */
  __CPROVER_cover(__sc_cbmc_covered_1); // Add assert for label 1 using its flag
  __CPROVER_cover(__sc_cbmc_covered_2); // Add assert for label 2 using its flag
  ...
}
```

its negation. If CBMC proves this assertion, then the label is unreachable (under the current assumptions). Otherwise, the counter-example provided by CBMC is exactly a test reaching the label condition. As each assertion is treated separately, this mode makes no effort on minimizing the number of tests generated. It will generate one test for each label. The harness used is similar to the one for the cover mode, except that the calls `__CPROVER_cover(cond)` are replaced with `__CPROVER_assert(!cond)`.

CBMC is a bounded model-checker. Therefore, it makes a bounded analysis of the program by unwinding loops a fixed number of times. In many cases, CBMC is able to determine automatically an upper bound on the number of loop iterations, but if the number of loop iterations highly depends on the data or is unbounded, the bound has to be specified by the user. For this, CBMC provides the option `--unwind n`, where `n` is the maximal number of times any loop condition should be evaluated. The consequence of this bounded analysis is that it is not necessarily complete, if the bound is not sufficient to represent all the initial program paths. This does not affect test generation, but the conclusion of uncoverability for a test objective is conditioned by the completeness of the analysis.

SeaCoral tries to lighten the burden of the user by minimizing the configuration needs, yet since this unwinding parameter cannot be automatically computed, it must be carefully set to obtain meaningful results when using CBMC through SeaCoral. This is one of the very few parameters of SeaCoral that need a bit of understanding of the code under test.

CBMC results are precise and allow to detect uncoverable labels efficiently. Its main drawback comes from its lack of collaboration from and to other analyzers.

Listing 1.2: Label implementation used in harnesses for KLEE.

```
# define cov_label(expr, id, ...)        \
  do {                                    \
    if (NONDET (id)) {                    \ /* Go to the label or bypass it.  */
      if (!__sc_is_covered (id))          \ /* If label id not yet covered,   */
        klee_assert (!(expr));            \ /* try to cover it; in any case   */
      klee_silent_exit (0);               \ /* exit without going further.    */
    }                                     \ /* Continue if label was bypassed.*/
  } while (0)
```

For instance, it cannot update the shared store during its execution: SeaCoral must wait until CBMC terminates the function's analysis to collect its results, extract test inputs, and submit them to the validator.

KLEE Integration. The integration of KLEE [15] is based on previous work [13]. KLEE symbolically executes the program path by path, and generates a test whenever it reaches a return statement of the main function, an assertion `klee_assert(0)`, or an instruction raising a runtime error like a division by zero or an illegal pointer access.

In the same way as we use `__CPROVER_assert(!cond)` to translate labels to targets for CBMC reachability analysis, we use `klee_assert(!cond)` to transform labels into targets for KLEE's test generation. Several optimizations are necessary to control the number and the length of the paths to explore, and to avoid generating tests for already covered labels (see details in [13]). We end up with the label instrumentation shown in Listing 1.2.

The `NONDET` call separates the paths into two separate branches: one continuing the execution, and one going into a branch in which the label's condition is tested. In both cases, KLEE's exploration is halted in order to cut off the branching introduced by the label: the first calls `klee_assert`, stopping the execution and returning a test; the second calls `klee_silent_exit`, artificially stating that the path is irrelevant and does not need further exploration. Cutting these branches solves both the path explosion and the growing complexity of the path conditions.

SeaCoral monitors tests generated by KLEE and feeds them to the validator while KLEE is running. Therefore, by updating the store, the execution of tests by the validator plays the role of a *concrete replay* that helps KLEE in avoiding exploration of already covered labels.

KLEE terminates once it finishes the exploration of the whole program, or until a given timeout, whichever happens first. In addition to taking benefit from the store state before and during the analysis, KLEE can be initialized on startup by seeding with tests from the corpus (T_{cov}) from which to start symbolic exploration.

KLEE could be configured to ensure that generated tests do not trigger RTEs by checking, like for CBMC's cover mode, that a label is actually covered *after* the call to the entrypoint function has terminated. Doing this check through the validator actually proved efficient and more reliable, as we found that KLEE's

Listing 1.3: Label instrumentation for fuzzing.

```
#define cov_label(expr, id, ...)         \
  do {                                   \
    if (expr) __asm__ volatile ("");     \ /* If expr holds, add an empty branch */
  } while (0)
```

error detection is not as precise as that of LLVM sanitizers (especially regarding dynamic memory).

libFuzzer Integration. libFuzzer [37] is the representative of *purely dynamic analyzers* that we have integrated in SeaCoral. To be efficient, a fuzzer needs to operate on a non-empty fuzzing corpus. Therefore, a first test sampling and retrieval phase constructs a fuzzing corpus that consists in a non-empty set of seeds. This is achieved by:

– re-using every *label-covering* test that is already recorded in SeaCoral's corpus;
– random sampling a given number of additional tests.

Then, a seed triage stage filters out randomly generated tests that trigger RTEs. This triage is necessary to determine whether the fuzzer has any valid seed left, i.e., a test that terminates, to be able to start its execution. This step is to prevent libFuzzer from entering an infinite loop trying to run tests that do not contribute any coverage (i.e., that do not lead to any increment in libFuzzer's internal counters) without any meaningful progress.

libFuzzer runs for a fixed period of time, or a given number of fuzzing iterations. A light instrumentation of labels is used to provide the fuzzer's internal constant monitor with meaningful values for mutations. We show this instrumentation in Listing 1.3, where `__asm__ volatile ("")` is an empty basic block that cannot be optimized out by C compilers: in this way, the branch condition is evaluated by libFuzzer, enabling it to derive meaningful constants.

SeaCoral dynamically retrieves label-covering tests that are recorded into SeaCoral's corpus by other analyzers while libFuzzer is running. It additionally submits tests to the validator as soon as they are persisted by the fuzzer.

LUncov Integration. LUncov is part of the LTest toolset [5, 8]. As such, it is already integrated into SeaCoral. It efficiently detects uncoverable labels from the results of Frama-C analyzes [6, 30], and more precisely of two of its plugins: Eva [14], an abstract interpretation engine working with various abstract domains, and WP [10], an analysis tool based on the computation of weakest preconditions.

6 Illustrative Examples and Benefits of SeaCoral

This section illustrates the benefits of using SeaCoral through additional examples. Since the analyzers integrated in the toolset are well-known and their capacities have already been demonstrated in previous work (and are preserved in SeaCoral), we focus on the new features enabled by the toolset.

```
void test_20 () {
  unsigned char (* buffer) = malloc (sizeof (unsigned char[4]));
  (void) memcpy (buffer, (unsigned char[4]){240, 128, 192, 255},
                 sizeof (unsigned char[4]));
  int n = 4;
  (void) check (buffer, n);
}
```

Fig. 3: Example of a label-covering test generated for function `check`

Check UTF-8 encoding. The `checkutf8` function (76 loc) checks whether a buffer of characters is a valid UTF-8 encoding (i.e. it translates to a sequence of valid UTF-8 encoded characters) and has the following signature:

```
int checkutf8 (unsigned char* buffer, int n)
```

We use SeaCoral with KLEE and the condition coverage criterion (CC) to cover each truth value of each subcondition of multiple conditions.

In order to avoid RTEs due to the inconsistency between the size of `buffer` and the value of `n`, SeaCoral offers the option `--array-size-mapping` to constrain the size of an array to equal the value of another parameter.

```
> seacoral --files checkutf8.c --entrypoint checkutf8 --criterion CC --tools klee
           --array-size-mapping buffer:n --max-ptr-array-length 10
```

With this configuration and precondition, SeaCoral launched with KLEE covers 100% of the test objectives required by the CC criterion (i.e., 52 labels) with 21 tests and finds no RTE, as expected. Fig. 3 shows an example of a label-covering test.

Array of queues. We consider a data structure of an array of queues, inspired by industrial code, with its initialization and update functions. A global variable `state` is declared as an array of 5 queues. The `initialize` function sets all pointers of queues to `NULL` and their sizes to 0. The `add` function adds a value at the end of the queue at a given index, while `pop` removes the first value of the queue at the given index if possible and stores the result in its other argument.

```
typedef struct node_t { int data; struct node_t *next; } node;
typedef struct queue_t { node *first; node *last; int length; } queue;

queue state[5];

void initialize();
void add(int val, int ind);
int pop(int ind, int *val);
```

We want to generate tests for the `pop` function with the condition coverage criterion CC.

```
> seacoral --files queue.c --entrypoint pop --criterion CC --tools klee
```

Without any additional configuration,SeaCoral launched with KLEE on function `pop` covers only 5 among the 8 test objectives for CC and finds 1 RTE. Unsurprisingly, the queues of the array `state` do not satisfy the implicit invariants on the length of the queues nor on the consistency of pointers. Furthermore, KLEE does not manage to cover the difficult parts of the function where there is actually a value to pop. The same goes if we try CBMC instead of KLEE.

The problem here is to generate meaningful states that allow to cover specific parts of the code. In such cases, the developer is willing to provide a test initialization function that builds a valid state from which relevant tests can be generated. For this, SeaCoral provides two options: `--ignore-globals state` makes SeaCoral ignore the global variable `state` that is too hard to symbolize due to its invariants; and `--fixtures-init init_valid_state` defines the hand-written function `init_valid_state` as a preamble for test case generation. Here we write this initialization function to create `state` with two queues, of size 1 and 2 respectively. With this configuration, both KLEE and CBMC manage to cover 100% test objectives for CC with 5 tests in less than 3 seconds. KLEE moreover finds 2 RTEs, when the pointer receiving the value is `NULL` or allocated with no memory.

PKCS#1 signature parser. PKCS#1 (Public-Key Cryptography Standard) specifies the usage of RSA algorithm, in particular for creation and verification of signatures. During PKCS#1 v1.5 signature verification, due to a complex structure of the signed message, a dedicated parser must be applied to extract necessary data. The implementation of this parser we target consists of 220 lines of code. Parsers are known to be difficult programs to cover by automatically generated tests, because of the complexity of the input structure and the numerous dependency constraints on the different fields composing it.

The signature of the entrypoint is the following:

```
int pars_PKCS1_lev1(uchar* pad_msg, int pad_msg_sz, uchar* in_msg, int in_msg_sz,
    uint expect_hash_ind)
```

We launch SeaCoral with CBMC with the decision coverage criterion (DC) and the following configuration: we constrain the size of arrays with their size arguments, the size of the largest input to 96 and the size of the largest loop to unfold to 65. These two last values are quite easy to determine by a quick analysis of the specification and the code. Moreover, with `sc_assume` we constrain the values of `expect_hash_ind` between 0 and 3 and the values of `in_msg_sz` and `pas_msg_sz` to be non-negative.

```
> seacoral --files pkcs1.c --entrypoint pars_PKCS1_lev1 --criterion DC
          --tools cbmc --unwind 65 --max-ptr-array-length 96
          --array-size-mapping pad_msg:pad_msg_sz,in_msg:in_msg_sz
```

With this configuration, CBMC runs 17 minutes and covers all 48 DC labels with 20 tests.

Benefits. These examples highlight the advantages of using SeaCoral. It requires a minimal configuration compared to that required by any of the analyzers used alone. The user does not need to create manually a harness to run any of these analyzers: SeaCoral generates it automatically, with test inputs made symbolic, even when their structure is complex and involves pointers. While all the analyzers generate values for test inputs in a tool-dependent format, the tests generated by SeaCoral are uniformly written (currently, in a C file), and can easily be maintained and replayed for regression checking.

7 Conclusion and Future Work

Industrial use of advanced testing tools based on formal methods still remains limited today. To address this issue, we propose SeaCoral, a new open-source toolset for C program testing, offering a complete range of testing services: annotation of test objectives for a given test criterion, test generation, identification of infeasible objectives and test replay. It provides a convenient framework for applying various tools, and currently integrates a model-checker, a fuzzer, a DSE tool and a static analyzer.

SeaCoral is currently being evaluated with success by Thales on a confidential industrial code of several thousands of lines, containing complex nested data structures and invariants and requiring test initialization functions to test state machine based behaviors.

Ongoing and future work include the maturation of the toolset and its evaluation on other industrial codebases. A visualization of the reached coverage, uncovered and uncoverable test objectives will help the user to understand SeaCoral's results and to find a better configuration. In some cases, user-provided annotations could be used to enrich the chosen criteria and force the generation of a test for a particular statement or branch. A support of hand-written test oracles will complete the coverage report by providing success or failure verdict for the generated tests. An integration of a runtime verification tool (e.g. E-ACSL [11]) could also complete the testing process with formal oracles executed at runtime. Parallel execution of several tools is already possible in SeaCoral and should be further evaluated on codebases of significant size and complexity.

Acknowledgment. Part of this work was funded by the French AID (Defence Innovation Agency) via the RAPID project AutoCouv and by ANR (grants ANR-22-CE39-0014, ANR-24-ASM2-0001). We thank the anonymous referees for their comments and suggestions.

References

[1] AFL: American Fuzzy Lop (Accessed Oct 16, 2025), URL `http://lcamtuf.coredump.cx/afl`

[2] Agrawal, H., DeMillo, R.A., Spafford, E.H.: Debugging with dynamic slicing and backtracking. Softw., Pract. Exper. **23**(6), 589–616 (1993), https://doi.org/10.1002/spe.4380230603

[3] Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC v4: Improving code coverage with smart seeds via BMC, fuzzing and static analysis. Formal Aspects Comput. **36**(2) (2024), https://doi.org/10.1145/3665337

[4] Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, 1 edn. (2008), ISBN 0521880386, 9780521880381

[5] Bardin, S., Chebaro, O., Delahaye, M., Kosmatov, N.: An all-in-one toolkit for automated white-box testing. In: Proc. of the 8th Int. Conf. on Tests and Proofs (TAP 2014), Held as Part of STAF 2014, LNCS, vol. 8570, pp. 53–60, Springer (2014), https://doi.org/10.1007/978-3-319-09099-3_4

[6] Bardin, S., Delahaye, M., David, R., Kosmatov, N., Papadakis, M., Traon, Y.L., Marion, J.: Sound and quasi-complete detection of infeasible test requirements. In: Proc. of the 8th IEEE Int. Conf. on Software Testing, Verification and Validation (ICST 2015), pp. 1–10, IEEE (2015), https://doi.org/10.1109/ICST.2015.7102607

[7] Bardin, S., Kosmatov, N., Cheynier, F.: Efficient leveraging of symbolic execution to advanced coverage criteria. In: Proc. of the 7th IEEE Int. Conf. on Software Testing, Verification and Validation (ICST 2014), pp. 173–182, IEEE (2014), https://doi.org/10.1109/ICST.2014.30

[8] Bardin, S., Kosmatov, N., Marcozzi, M., Delahaye, M.: Specify and measure, cover and reveal: A unified framework for automated test generation. Sci. Comput. Program. **207** (2021), https://doi.org/10.1016/j.scico.2021.102641

[9] Bardin, S., Kosmatov, N., Marre, B., Mentré, D., Williams, N.: Test case generation with PathCrawler/LTest: How to automate an industrial testing process. In: Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'18), LNCS, vol. 11247, pp. 104–120, Springer (2018), https://doi.org/10.1007/978-3-030-03427-6_12

[10] Baudin, P., Bobot, F., Correnson, L., Dargaye, Z., Blanchard, A.: WP Plug-in Manual (2023), URL `https://frama-c.com/download/frama-c-wp-manual.pdf`

[11] Benjamin, T., Signoles, J.: Runtime Annotation Checking with Frama-C: The E-ACSL Plug-in. In: Kosmatov, N., Prevosto, V., Signoles, J. (eds.) Guide to Software Verification with Frama-C: Core Components, Usages, and Applications, pp. 263–303, Computer Science Foundations and Applied Logic Book Series, Springer (2024), https://doi.org/10.1007/978-3-031-55608-1_5

[12] Berthier, N., de Oliveira, S., Kosmatov, N., Longuet, D.: SeaCoral: A Collaborative Test Generation Toolset. Companion Artifact (2025), https://doi.org/10.5281/zenodo.17357288

[13] Berthier, N., de Oliveira, S., Kosmatov, N., Longuet, D., Soulat, R.: An efficient black-box support of advanced coverage criteria for Klee. In: Proc. of the 38th Annual ACM/SIGAPP Symp. on Applied Computing, Software Verification and Testing Track (SAC-SVT 2023), pp. 1706–1715, ACM (2023), https://doi.org/10.1145/3555776.3577713

[14] Bühler, D., Maroneze, A., Perrelle, V.: Abstract interpretation with the Eva plug-in. In: Kosmatov, N., Prevosto, V., Signoles, J. (eds.) Guide to Software Verification with Frama-C: Core Components, Usages, and Applications, pp. 131–186, Computer Science Foundations and Applied Logic Book Series, Springer (2024), https://doi.org/10.1007/978-3-031-55608-1_3

[15] Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI

2008), pp. 209–224, USENIX Association (2008), URL `http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf`

[16] Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: Proc. of the 33rd Int. Conf. on Software Engineering (ICSE 2011), pp. 1066–1071, ACM (2011), https://doi.org/10.1145/1985793.1985995

[17] Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013), https://doi.org/10.1145/2408776.2408795

[18] Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. Softw. Eng. J. **9**(5), 193–200 (1994), https://doi.org/10.1049/SEJ.1994.0025

[19] Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. of the Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), LNCS, vol. 2988, pp. 168–176, Springer (2004), https://doi.org/10.1007/978-3-540-24730-2_15

[20] Collavizza, H., Vinh, N., Ponsini, O., Rueher, M., Rollet, A.: Constraint-based BMC: A backjumping strategy. Int. Journal on Software Tools for Technology Transfer **16**(1), 103–121 (2014), https://doi.org/10.1007/s10009-012-0258-6

[21] Fioraldi, A., Mantovani, A., Maier, D.C., Balzarotti, D.: Dissecting American Fuzzy Lop: A FuzzBench Evaluation. ACM Trans. Softw. Eng. Methodol. **32**(2), 52:1–52:26 (2023), https://doi.org/10.1145/3580596

[22] Gigante, G., Pascarella, D.: Formal methods in avionic software certification: The DO-178C perspective. In: Proc. of the 5th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies (ISoLA 2012), LNCS, vol. 7610, pp. 205–215, Springer (2012), https://doi.org/10.1007/978-3-642-34032-1_21

[23] Goodman, P., Groce, A.: DeepState: A unit test-like interface for fuzzing and symbolic execution (2018), URL `https://github.com/trailofbits/deepstate`

[24] Google: Documentation for OSS-Fuzz (2025), URL `https://google.github.io/oss-fuzz/`

[25] ISO Standard: ISO 26262-10:2018 Road vehicles — Functional safety (2018), URL `https://www.iso.org/standard/68392.html`

[26] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. pp. 1–37 (2015), https://doi.org/10.1007/s00165-014-0326-7

[27] Laski, J.W., Korel, B.: A data flow oriented program testing strategy. IEEE Trans. Software Eng. **9**(3), 347–354 (1983), https://doi.org/10.1109/TSE.1983.236871

[28] Lavillonnière, E., Mentré, D., Cousineau, D.: Fast, automatic, and nearly complete structural unit-test generation combining genetic algorithms and formal methods. In: Proc. of the 13th Int. Conf. on Tests and Proofs (TAP 2019), LNCS, vol. 11823, pp. 55–63, Springer (2019), https://doi.org/10.1007/978-3-030-31157-5_4

[29] Liu, Y., Deng, J., Jia, X., Wang, Y., Wang, M., Huang, L., Wei, T., Su, P.: PromeFuzz: A Knowledge-Driven Approach to Fuzzing Harness Generation with Large Language Models. In: Proc. of the 2025 ACM SIGSAC Conf. on Computer and Communications Security, pp. 1559–1573 (2025), https://doi.org/10.1145/3719027.3765222

[30] Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L.: Time to clean your test objectives. In: Proc. of the 40th Int. Conf. on Software Engineering (ICSE 2018), pp. 456–467, ACM (2018), https://doi.org/10.1145/3180155.3180191

[31] Mathur, A.P.: Foundations of Software Testing. Addison-Wesley Professional (2008)

[32] Metta, R., Medicherla, R.K., Chakraborty, S.: Bmc+fuzz: Efficient and effective test generation. In: Proc. of the 2022 Conf. & Exhibition on Design, Automation & Test in Europe (DATE 2022), pp. 1419–1424 (2022), https://doi.org/10.23919/DATE54114.2022.9774672

[33] Microsoft: OneFuzz: A self-hosted fuzzing-as-a-service platform (2023), URL `https://github.com/microsoft/onefuzz`

[34] Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley, 3 edn. (2011)

[35] Poeplau, S., Francillon, A.: Symbolic execution with SymCC: Don't interpret, compile! In: Proc. of the 29th USENIX Security Symp., pp. 181–198, IEEE (2020), https://doi.org/10.5555/3489212.3489223

[36] SeaCoral: The SeaCoral Testing Toolset (Accessed Oct 16, 2025), URL `https://github.com/ocamlpro/seacoral`

[37] Serebryany, K.: Continuous Fuzzing with libFuzzer and AddressSanitizer. In: Proc. of the IEEE Int. Conf. on Cybersecurity Development, (SecDev 2016), p. 157, IEEE (2016), https://doi.org/10.1109/SECDEV.2016.043

[38] TestComp: Int. competition on software testing (2026), URL `https://test-comp.sosy-lab.org/`

[39] Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: Proc. of the 5th European Dependable Computing Conf. (EDCC 2005), LNCS, vol. 3463, pp. 281–292, Springer (2005), https://doi.org/10.1007/11408901_21

[40] Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: The Fuzzing Book. CISPA Helmholtz Center for Information Security (2024), URL `https://www.fuzzingbook.org/`

[41] Zhang, G., Shuai, Z., Ma, K., Liu, K., Chen, Z., Wang, J.: Fdse: Enhance symbolic execution by fuzzing-based pre-analysis (competition contribution). In: Proc. of the Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2024), LNCS, vol. 14573, pp. 304–308, Springer (2024), https://doi.org/10.1007/978-3-031-57259-3_16

[42] Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**(4), 366–427 (1997), https://doi.org/10.1145/267580.267590