An Efficient Black-Box Support of Advanced Coverage Criteria for Klee

Nicolas Berthier
Steven De Oliveira
OCamlPro
Paris, France
nicolas.berthier@ocamlpro.com
steven.de-oliveira@ocamlpro.com

Nikolai Kosmatov, Delphine Longuet, Romain Soulat
Thales Research & Technology
Palaiseau, France
nikolai.kosmatov@thalesgroup.com
delphine.longuet@thalesgroup.com
romain.soulat@thalesgroup.com

ABSTRACT

Dynamic symbolic execution (DSE) is a powerful test generation approach based on an exploration of the path space of the program under test. Well-adapted for path coverage, this approach is however less efficient for conditions, decisions, advanced coverage criteria (such as multiple conditions, weak mutations, boundary testing) or user-provided test objectives. While theoretical solutions to adapt DSE to a large set of criteria have been proposed, they have never been integrated into publicly available testing tools. This paper presents a first integration of an optimized test generation strategy for advanced coverage criteria into a popular open-source testing tool based on DSE, namely, Klee. The integration is performed in a fully black-box manner, and can therefore inspire an easy integration into other similar tools. The resulting version of the tool, named Klee4labels, is publicly available. We present the design of the proposed technique and evaluate it on several benchmarks. Our results confirm the benefits of the proposed tool for advanced coverage criteria.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; **Software testing and debugging**.

KEYWORDS

Test generation, dynamic symbolic execution, test coverage criteria, Klee, coverage labels.

ACM Reference Format:
Nicolas Berthier, Steven De Oliveira, and Nikolai Kosmatov, Delphine Longuet, Romain Soulat. 2023. An Efficient Black-Box Support of Advanced Coverage Criteria for Klee. In *Proceedings of ACM SAC Conference (SAC'23).* ACM, New York, NY, USA, Article 12927.130, 10 pages. https://doi.org/10.1145/3555776.3577713

1 INTRODUCTION

Automatic test generation techniques have made significant progress during the past two decades. One of the most remarkable successes in that area is *dynamic symbolic execution* (DSE) [8, 9], an efficient test generation technique combining symbolic and concrete executions of the program under test. Several efficient DSE tools have been developed [6, 7, 13, 15, 28–30, 30, 31], and several case studies and industrial applications have been reported [4, 6, 7, 15].

Dynamic symbolic execution relies on an exhaustive exploration of the path space of the program under test. Depending on the tool, several strategies can be available: depth-first search, breadth-first search, as well as more elaborated heuristics. While DSE is well-adapted for path coverage, it is less suitable for other coverage criteria, such as conditions, decisions, multiple conditions, weak mutations, boundary testing, or user-provided test objectives. Indeed, various coverage criteria require to cover quite different test objectives in the program code, and the support of different criteria in the existing tools remains limited. While the all-path criterion is powerful, it is sometimes too expensive when only decisions or conditions need to be covered. And the other way round, the tests generated for the all-path criterion may miss interesting behaviors with respect to such criteria as multiple conditions, weak mutations, or limit values. In practice, validation engineers aim at satisfying a given coverage criterion with a test suite of manageable size.

Bardin et al. [2] proposed a framework for specifying coverage criteria in a unified way with elementary test objectives, called *(coverage) labels*, and proposed theoretical solutions for an efficient test generation for labels. However, these techniques have never been integrated into any publicly available DSE tool. The only known implementation of these techniques is a greybox integration [2] inside PathCrawler [31], a proprietary test generation tool, whose industrial evaluation was reported to be quite efficient [4].

The use of labels to represent coverage criteria allows for a very generic approach to criteria-guided test generation. Labels provide a lightweight solution to encode various criteria, which is entirely independent of the underlying test generation tool. Furthermore, many basic and advanced coverage criteria can be represented using labels, and even custom labels can be added by hand when specific test objectives are needed. The absence of a publicly available DSE tool with an efficient support for a large range of coverage criteria remains a serious barrier for a larger support of coverage criteria. Addressing this issue is the main motivation of this work.

Our goal is to demonstrate how a dedicated support for labels can be integrated into Klee [6], a popular open-source test generation tool based on DSE. Contrary to the earlier greybox integration into PathCrawler [2], whose path exploration strategy had to be strongly modified and adapted for labels, we perform a fully black-box integration, that can inspire a lightweight integration into other similar tools. The resulting version of the tool, named Klee4labels, is publicly available. Finally, we evaluate the proposed version on several benchmarks. Overall, those benchmarks demonstrate the ability of the proposed tool: (i) to drive symbolic execution towards the test objectives required by a given coverage criterion, (ii) to generate fewer and more meaningful test cases, (iii) with a reasonable overhead compared to the original version of the tool.

Contributions. The contributions of this work include:

- a lightweight black-box integration of an efficient support for various coverage criteria expressed by labels in a popular publicly available tool Klee, making it possible to support a large panel of coverage criteria;
- a detailed description of the integration as a generic approach that is expected to be reproducible with other similar tools;
- an evaluation of the extended version of Klee on several benchmarks, and a detailed analysis of their results in comparison with other approaches, confirming the benefits of the proposed technique.

Outline. Section 2 presents necessary background on dynamic symbolic execution, the Klee tool, and on expressing test coverage criteria with coverage labels. Some motivating examples are given in Section 3. The design of our optimized test generation technique for Klee is presented in Section 4. Section 5 provides a detailed evaluation of the proposed technique in comparison to other approaches for dealing with labels. Related work is discussed in Section 6. Finally, Section 7 gives a conclusion and some directions for future work.

2 BACKGROUND

2.1 Dynamic Symbolic Execution

Symbolic execution [9, 17] is a powerful approach to automatic test generation, based on systematic path exploration. It consists in computing, for each path of the program under test, its *path condition*, that is, the set of conditions on the program parameters that must hold to ensure that the program executes along this path (under the assumption that the program is deterministic). If this path condition is satisfiable, then a solution gives concrete values to the parameters, and thus a test case that executes this path. If it is not satisfiable, then the path is called infeasible, meaning that it does not correspond to any possible execution of the program. In practice, the path may contain statements about which it is difficult to reason symbolically, for example, calls to external functions, which can lead to an under-approximation of the path constraint. Dynamic symbolic execution [8] interleaves concrete and symbolic execution and uses the gathered information to better approximate path constraints.

2.2 Klee

Klee [6] is a popular test-case generation tool for C programs based on dynamic symbolic execution, developed and maintained at Imperial College London. Klee is open-source, has a large community of contributors, and a large user base, both in industry and in academia. Klee operates on LLVM bitcode, which is an intermediate representation of the executable program: this enables it to mix concrete and symbolic executions. Klee internally makes use of various (configurable) strategies to explore the path space of the program under test, and is able to produce a test case for each path found to be feasible. Such a test case consists of concrete input values on which the program executes along this path. By design, Klee's only coverage criterion is *all-path*: as a result, the user often must configure a timeout or specify preconditions to the program in order to ensure Klee's termination within reasonable time bounds. More precisely, Klee aims at covering all paths of the LLVM bitcode, which means in particular that compound conditions are decomposed according to lazy evaluation semantic of Boolean operators in C.

Concretely, when launched on a given program, Klee tries to produce a test case for each executable path that reaches a return statement of the main function, an assertion, or an instruction that may raise a runtime error (RTE), e.g. division by 0, overshift, invalid pointer access. A path that reaches a return statement is called a *complete path*, whereas other paths are called *partially completed paths*. In Klee's output directory, a test case generated for a partially completed path comes with an additional file that ends with `.`*xxx*`.err`, where *xxx* gives information about the premature end of the path, e.g. `assert` (for assertion failure), or `ptr` (for pointer error). Klee can replay generated test cases in a separate step.

Since Klee aims at covering all paths of the LLVM bitcode, it sometimes produces more test cases than needed to cover only decisions or conditions for example. Its path-oriented approach is not directly adapted to more advanced coverage criteria like multiple conditions, boundary tests or weak mutations (cf. Section 2.3). The purpose of this work is to adapt and optimize Klee for a large range of coverage criteria expressed using coverage labels.

2.3 Coverage Labels

Bardin et al. [2] introduce a generic approach to represent coverage criteria as source code annotations. They propose to represent the test objectives required to be covered to satisfy a given coverage criterion as a set of annotations named *coverage labels* (that we often abbreviate as *labels*). A coverage label ℓ is defined as a pair loc, p, where p is a predicate attached to some program location loc. Such a label is covered by a test if the execution of this test reaches the location loc and satisfies the predicate p at this location. In this way, when the underlying test generation tool produces test cases covering all labels corresponding to a given coverage criterion, it builds a test suite satisfying the corresponding criterion.

Previous work proposed LTest [3], a toolset dedicated to labels. It is developed as a set of plugins of Frama-C [18], a verification platform for C code. The LTest toolset notably comprises the LAnnotate tool that, given a C program and a coverage criterion, automatically annotates that program by adding the corresponding labels. More precisely, given a coverage criterion $\mathscr{C}$ and the C source code of a program P, it automatically outputs a C program P' with additional coverage labels with respect to $\mathscr{C}$, i.e. such that a test suite satisfies $\mathscr{C}$ iff it covers all the added labels.

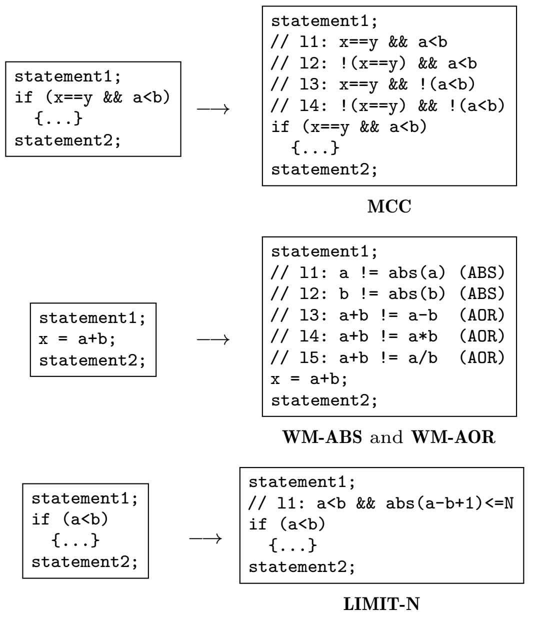
Basic coverage criteria can be simulated by coverage labels: for example, instruction coverage (**IC**), decision coverage (**DC**) or condition coverage (**CC**, where all atomic conditions of each decision, and their negations, must be covered). Each label encodes an elementary test objective required to be covered.

Coverage labels can additionally encode more advanced criteria on conditions, boundary (that is, limit) values or mutations. Multiple condition coverage (**MCC**) requires to cover all combinations of truth values for all atomic conditions of each decision. For a chosen set of mutation operators, weak mutation coverage (**WM**) requires to cover (or, as it is often called, to *kill*) each mutant program. A mutant is killed by a test execution if the mutation point is reached and, after the mutated instruction, some variable has a different value in the mutant compared to the original program. This can typically be encoded by a label with a predicate ensuring non-equality between the original and mutated expressions. If such a label is covered by a test, this test kills the corresponding mutant (in the aforementioned sense). Many common weak mutation operators, such as ABS (absolute value insertion), ROR (relational operator replacement), AOR (arithmetic operator replacement), and COR (logical operator replacement), can be simulated with labels.

Boundary testing requires to cover limit values of variables or conditions. For example, in the condition limit coverage criterion, for a condition `a<b` with two integer variables `a` and `b`, the boundary test objective is `a-b+1==0`. Since the exact boundary value can be unreachable, test engineers may want to get sufficiently close to the boundary, up to a chosen distance value N. This is the purpose of the criterion **LIMIT-N** of LAnnotate. For example, for a condition `a<b`, this requires to cover the boundary test objective `abs(a-b+1)<=N`.

Figure 1 shows label annotation for **MCC**, for weak mutations **WM-ABS** and **WM-AOR** with mutation operators ABS and AOR, and for **LIMIT-N**. Finally, custom labels can be added by test engineers when specific test objectives are needed.

Test generation for labels. Once coverage labels have been inserted into the program under test, they are transformed in order to be used by the underlying DSE test generation tool. Such a transformation (typically realized by program instrumentation) determines a modified path exploration strategy



```
statement1;
if (x==y && a<b)
  {...}
statement2;
```
⟶
```
statement1;
// l1: x==y && a<b
// l2: !(x==y) && a<b
// l3: x==y && !(a<b)
// l4: !(x==y) && !(a<b)
if (x==y && a<b)
  {...}
statement2;
```

```
statement1;
x = a+b;
statement2;
```
⟶
```
statement1;
// l1: a != abs(a) (ABS)
// l2: b != abs(b) (ABS)
// l3: a+b != a-b  (AOR)
// l4: a+b != a*b  (AOR)
// l5: a+b != a/b  (AOR)
x = a+b;
statement2;
```

```
statement1;
if (a<b)
  {...}
statement2;
```
⟶
```
statement1;
// l1: a<b && abs(a-b+1)<=N
if (a<b)
  {...}
statement2;
```

Figure 1: Label annotation for some coverage criteria.

and therefore represents a key element of the resulting test generation technique.

Bardin et al. [2] propose theoretical solutions to optimize test generation for labels. First, they present two instrumentations, called *direct* and *tight*. Given a label $\ell = loc, p$ with location loc and predicate p, direct instrumentation replaces ℓ with a new branching instruction `if (`p`) {}`. Tight instrumentation replaces ℓ with a non-deterministic choice leading to a new assertion on p and an exit: `if (nondet){ assert (`p`); exit (0); }`.

Direct instrumentation is shown by Bardin et al. [2] to exponentially increase the number of paths in the instrumented program, contrary to the tight one. Indeed, with direct instrumentation, for each path traversing a label location loc in the initial program, there are two paths going through loc (with p being either true or false) in the instrumented program. This multiplication of paths becomes even more significant when some paths in the initial program traverse location loc several times (e.g. because of loops or function calls). As a consequence, a DSE tool generates a test for each path traversing a label location, even if this label is already covered by a previously generated test. To reduce such redundancy in test generation, a first proposed optimization is tight instrumentation. For each path leading to a label location, tight instrumentation adds a unique path that leads to the label condition verification and exits the program immediately after that, otherwise the program ignores the label and continues.

The idea of the second optimization, called *iterative label deletion* (ILD), is to replay tests as soon as they are generated to mark all labels that are covered during test execution. This is used to prevent test generation from attempting to cover those labels anymore. Indeed, the execution of a test generated to cover a given label ℓ may cover other labels, for which it is no longer necessary to generate a test. Tight

Listing 1: `power.c`

```
int power (int X, int N) {
  int S = 1, Y = X, P = N;
  while (P >= 1) {
    if (P % 2 != 0) {
      P = P - 1;
      S = S * Y;
    }
    Y = Y * Y;
    P = P / 2;
  }
  return S;
}
```

instrumentation and iterative label deletion are expected to lead to a very efficient way for dynamic symbolic execution to handle label coverage. This is confirmed by Bardin et al. [2] after a dedicated gray-box integration of these approaches into a proprietary tool PathCrawler [31]. Until now, however, these optimizations have never been integrated into a publicly available testing tool, nor have they been realized in a fully black-box manner. In this work, we propose to leverage the genericity of this approach to a renowned publicly available testing tool, and to do it in a fully black-box manner. In doing so, we effectively extend the label-agnostic test generation tool Klee to all coverage criteria that can be expressed with labels.

3 MOTIVATING EXAMPLES

The program `power` of Listing 1 takes two integer inputs `X` and `N` and computes $\texttt{X}^{\texttt{N}}$ when `N` is non-negative[1]. In this program, each path corresponds to exactly one value of `N`. For example, the path entering the loop twice, the first time with an even value for `P` and the second time with an odd value for `P`, only executes if `N=2`. When `N` is bounded between 0 and a maximum value B, with $B > 1$, Klee explores B 1 paths to always generate three tests[2]: one with `N=0`, one with an even value for `N` and one with an odd value for `N`. As one increases the value of B, the exploration time grows, even if only the three same tests are produced in the end.

For this kind of programs consisting of a very simple control flow with a large set of different paths, Klee spends a lot of time exploring all the paths: 8.3 s for $B = 100$, 52 s for $B = 1000$, 553 s for $B = 5000$, to give a few examples. However, only a few paths are necessary to cover basic criteria like instructions, decisions or conditions. We will show how the support of labels can drastically improve the exploration time of such programs while still producing a relevant test set of a small size.

As another example, consider the function `search` of Listing 2. For a fixed maximal length $L > 1$ of `tab`, and `n` assumed to be between 0 and L, Klee generates 3 tests covering all branches, for instance: (1) `n=0`, (2) `n=1`, `tab[0]=0` and `val=0`, (3) `n=2`, `tab[0]=42`, `tab[1]=0`, and `val=0`. We observe that decisions and conditions are covered, but the multiple condition `!res && i<n` is never evaluated with the combination `!res` and `!(i<n)`. Indeed, in the three tests generated, the searched value is found in the array. The execution exits the loop because `res=1` and not because the end of the array is reached. Therefore, the test suite generated by Klee does not satisfy multiple condition coverage.

[1]For simplicity, we ignore arithmetic overflows for this example.
[2]We use the option `-only-output-states-covering-new` so that Klee only generates test cases covering yet uncovered code.

Listing 2: `search.c`

```
int search (int *tab, int n, int val) {
  int res = 0, i = 0;
  while (!res && i < n) {
    if (tab[i] == val)
      res = 1;
    i++;
  }
  return res;
}
```

We observe that for coverage criteria that are not subsumed by all-path (like multiple conditions, weak mutations, or condition limits), the test suites generated by Klee may be incomplete. We will show how the support of labels helps to achieve a high coverage of such criteria with a reasonable overhead.

4 DESIGN OF AN OPTIMISED APPROACH

Before delineating our optimized approach for targeting labels in Klee, we first describe a naive approach and illustrate the overall test generation process. Along with informal notation for labels of the form *`// lid: expr`* used in the examples of Figure 1, we will use a macro `cov_label(expr, id)` to denote labels in annotated C programs, where `id` is a unique integer identifier associated with the label, and `expr` is the C expression of its predicate p. The macro is practical to define the required behavior, which can vary between the different test generation approaches and for test replay.

4.1 Naive Approach

The naive approach to the problem corresponds to direct instrumentation (cf. Section 2.3), where each label adds an additional branching instruction whose both branches lead back to the next statement. Instrumenting the program to achieve this consists in replacing each label `cov_label(expr, id)` by a new branching instruction `if (expr){}`, as illustrated in Figure 2. In this way, we effectively instruct Klee to try to generate at least one test case that covers the empty branch, i.e. where `expr` holds and the label is covered.

In practice, Klee currently does not handle empty code blocks (e.g. inserted by a statement that cannot be optimized out by a C compiler, like `__asm__ volatile ("");`). To circumvent this limitation, we implement the empty branch using a call to an external function `nop` that returns immediately[3].

Measuring Coverage. Executing Klee on the instrumented program produces a set of test cases T. Measuring the coverage of T w.r.t. a given criterion boils down to replaying each test case from T and recording all covered labels. We replay all generated test cases using a specific instrumentation of the

[3]This function is given in a separate library, and its implementation is therefore not subject to Klee's scrutiny: it is only called when Klee's concrete execution traverses this branch.

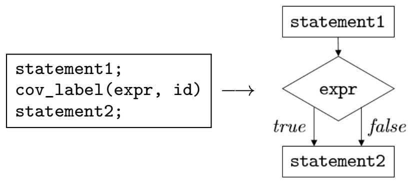


Figure 2: Naive (direct) instrumentation of labels

program under test. A coverage status for each label (covered or uncovered) is recorded in a shared, persistent store that we denote by σ.

More concretely, we construct an executable version of the program called the *replayer*, where each label `cov_label(expr, id)` is replaced with a call to a function `set_covered` guarded with `expr`, i.e. `if (expr){ set_covered (id); }`. The role of `set_covered (id);` is to register into σ that the label identified with `id` has been covered. After replaying every test case using the replayer, the coverage achieved by T is recorded in σ and can be measured.

Runtime Errors. Notice that, as mentioned in Section 2.2, one of Klee's goals is to produce test cases that trigger runtime errors (RTEs). Yet, we need to take special care regarding such test cases for the two following reasons. First, our objective is *not* to find test cases that trigger RTEs: instead, we want to achieve label coverage. If RTEs are detected, they should be reported, analyzed and fixed separately (e.g. by adjusting the preconditions or fixing possible bugs in the code). This task is outside the scope of this work. Second, RTEs actually correspond to *undefined* behaviors, hence, by nature, label coverage achieved after an undefined behavior cannot be considered as firmly achieved.

We have thus chosen to keep only tests generated for complete paths, and to ignore tests with an .err file (for assertions or RTEs). In practice, to correctly record label coverage, we redefine the role of the `set_covered` function used by the replayer, and make it insert new label statuses into a temporary buffer $\tilde{\sigma}$. Then, at the end of its execution, if it is successful (i.e. the test execution terminates normally), the replayer commits the contents of $\tilde{\sigma}$ into the store σ. In this way, when a test case t triggers an RTE *after* having covered a label ℓ, the replayer does not effectively update the status of ℓ in the persistent store σ, and the coverage induced by t is thus not accounted for: t is removed from T without compromising the consistency of the reported coverage measures. This gives us the set of test cases T', with $T' \subseteq T$.

4.2 Optimized Approach

While it is capable of achieving label coverage in theory, several drawbacks cripple the naive approach delineated above in practice. First, direct instrumentation induces an exponential explosion in the number of paths that must be explored. Second, the constraints (*path conditions*) that are accumulated during the symbolic execution of a path grow in complexity each time the path traverses the branching instruction added for a label. This can be observed in Figure 2, where every path reaching `statement2` traverses the guard `if (expr)`.

Tight Instrumentation. In their quest for a more efficient test case generation for label coverage, Bardin et al. [2] first designed tight instrumentation by observing that: (i) the expression `expr` in `cov_label(expr, id)` is only relevant to covering the label; and (ii) the expression `!expr` is irrelevant in any path. Tight instrumentation "cuts" the branch that covers a label, and prevents the propagation of `!expr` into path conditions for longer paths. To design a practically applicable version of the general idea of Bardin et al. [2], we guard the condition on `expr` for a label `cov_label(expr, id)` with a non-deterministic choice encoded using an additional symbolic variable whose *unique* name is built using `id`. Since it is never assigned anywhere in the program, this variable is effectively an additional input used to simulate the non-deterministic choice for the label: either the label is to be considered, or it is ignored.

Klee provides several primitives that allow us to achieve this. First, we can force the generation of a test case with expression `e` being null by using a statement `klee_assert (e);`. Second, a statement `klee_silent_exit (0);` stops any further exploration from its location, and does not generate any test case. Finally, `klee_int ("varname")` represents the value of a symbolic integer variable (of C type `int`) with name `varname`. As a result, tight instrumentation can be achieved by replacing any label `cov_label(expr, id)` with

```
  if (NONDET (id)) {
    klee_assert (! (expr));
    klee_silent_exit (0);
  }
```

where `NONDET(id)` defines a symbolic integer variable as `klee_int ("nondet_"TOSTRING (id))`. If the non-deterministic choice indicates that the label is to be considered (i.e. `NONDET(id)` holds), then Klee tries to generate a test with a failure of `klee_assert ( !(expr))` (that is, the label being covered). Otherwise the assert (and, therefore, the label) is ignored.

Contrary to the naive approach where the set of test cases T consists of test cases generated by Klee for complete paths, in the optimized approach we are only interested in keeping tests that violate a newly inserted assertion statement. We achieve this straightforwardly by only considering tests for which a file with an `.assert.err` extension exists. As a simple additional optimization, we insert a call `klee_silent_exit (0);` at the very end of the tested program: in this way, we instruct Klee to ignore every complete path that covers yet uncovered *code*, but covers no label.

Iterative Label Deletion. Iterative label deletion (ILD) consists in preventing symbolic execution from trying to cover a label that has already been covered. Our optimized approach implements ILD by making use of an *external* function `covered`[4] that, given the identifier `id` of a label, returns a non-null integer if the label has already been covered. This function simply queries the persistent store σ for the status of label `id`.

[4]Similarly to `nop`, the function `covered` is defined in a separate library.

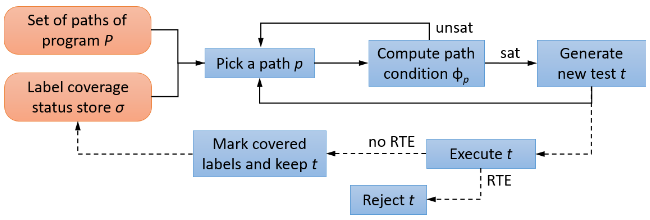

Set of paths of program P
Label coverage status store σ
Pick a path p
unsat
Compute path condition ϕ_p
sat
Generate new test t
Mark covered labels and keep t
no RTE
Execute t
RTE
Reject t

Figure 3: Iterative label deletion

Our implementation of ILD is illustrated by Figure 3, where solid arrows indicate the main test generation flow and dashed arrows show the replay steps performed in parallel. Note that our implementation of ILD is slightly different from the theoretical approach of Bardin et al. [2]. In fact, they propose to replay each test as soon as it is generated, while the path exploration is suspended and waits for the results of the replay before continuing. We do not record that a label is covered in σ as soon as the corresponding assertion fails on a test case, but only during its replay. Moreover, we only replay a test case when we detect that Klee outputs a new test file that leads to a failure of the assertion — i.e. the name of its associated file is suffixed with `.assert.err`. The path exploration of Klee continues in parallel in the meantime, as it is illustrated by the dashed lines in Figure 3. This means that the result of subsequent calls to `covered` may depend on the timing of system-level operations. An integration *à la* Bardin et al. (where the path exploration is suspended during the replay) may require altering the source code of Klee, to insert specific code to launch and wait for a replayer sub-process as soon as a test case is generated. This solution would delay the path exploration of Klee by extra waiting time and would not be compatible with our goal to perform a black-box integration into Klee.

Our combination of tight instrumentation and ILD (illustrated in Figure 4) is achieved by replacing each `cov_label(expr, id)` with:

```
if (NONDET (id)) {
  if (!covered (id))
    klee_assert (! (expr));
  klee_silent_exit (0);
}
```

By guarding the assertion with a call to `covered`, we cut any branch leading to an already covered label, and instruct Klee to ignore the potentially complex expression of the label predicate.

An attentive reader will notice that this version is slightly different from the proposal of Bardin et al. [2, Fig. 7] that was suboptimal. Indeed, in their proposal (and contrary to their declared intention), in the case where a non-deterministic choice indicates that the label must be considered but it is marked as already covered, the program does not exit and symbolic execution continues the exploration of the rest of the program. Hence the following branches can be explored in a redundant way, when a non-deterministic choice is true and false. This apparently small issue was rather tricky to find and to fix. This confirms that it is important to validate a theoretical approach by a working tool implementation.

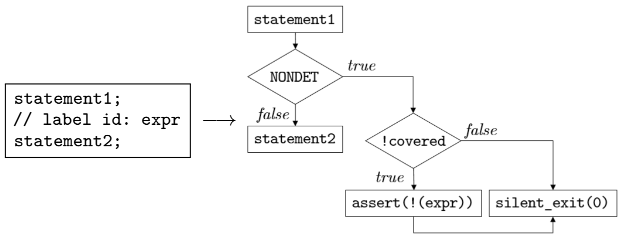


Figure 4: Optimized instrumentation of labels

Measuring Coverage. An essential feature of the optimized approach is that the code instrumented for the path exploration in Klee is not the same as for measuring the coverage of test cases. Thanks to the intrumentation presented above, during the path exploration, the condition of a label is considered only when necessary: at most once on a program path and only if the label is not yet covered. The coverage computation relies again on the replayer described in Section 4.1. Therefore, during the replay of a test case, all labels are checked for being covered.

5 EVALUATION

Thanks to adopting a black-box approach, the effort to implement our approach as a front-end for Klee is limited. Indeed, our tool prototype Klee4labels mostly consists of about 700 lines of OCaml code, along with about 300 lines of C for instrumentation macros and the library of external functions. This prototype is publicly available[5].

To help us in evaluating our approach, we have also designed an extended version of Klee4labels, with a more advanced implementation of the label coverage store (whose optimized implementation is proprietary), an automated generation of test harnesses, and further treatment of generated tests to produce readable C code for instance. Nevertheless, the main optimization feature is the instrumentation of labels, which is the same in both versions of the tool. The rest of this section shows evaluation results based on the extended version of Klee4labels. We consider the following research questions:

RQ1 **Label coverage**: does the support for labels lead to a high coverage of labels by Klee?

RQ2 **Size of the test suite**: does the label-guided generation lead to test suites of reasonable sizes?

RQ3 **Efficiency**: what is the time overhead that is due to the exploration of labels by Klee?

To answer these questions, we chose to evaluate our optimized front-end for Klee on the same programs as in [2], plus a few demonstrative toy examples: `tritype` is a well-known program widely used for test generation; `power` is the program presented in Section 3; `selection_sort` is a classic implementation of a sorting algorithm; `modulus` comes from TestComp'22 [5]; `fourballs` comes from [26]; `tcas` and `replace`

[5]https://github.com/OCamlPro/klee4labels hosts the full source code of Klee4labels, along with several Klee drivers that correspond to the examples of this section.

come from the Siemens test suite [12]; `get_tag` and `full_bad` come from the Verisec benchmark [20]; and `gd` comes from MediaBench [22]. Experiments were performed on a laptop computer with 16 GB of RAM and a 1.8 GHz Intel Core i7 CPU running GNU/Linux. We used the latest development version of Klee available[6] with its default options, on our benchmark programs compiled to LLVM bitcode using Debian clang version 14 (with default optimization settings). The results[7] are presented in Table 1.

To evaluate our tool in terms of label coverage, test suite size and execution time, we ran it on the same programs with four different modes: a mode without label instrumentation, and three modes supporting labels in different ways. In the first mode, called the *ignore* mode in Table 1, each label is replaced with an empty instruction block. Thus, this mode corresponds to what Klee does on its own, and is used as a witness. The second mode implements the *naive* instrumentation described in Section 4.1. The third mode uses *tight* instrumentation but does not replay test cases as soon as they are generated, thus performing a full exploration of the instrumented program. The last mode, called *optim*, corresponds to our optimized implementation, with tight label instrumentation and iterative label deletion, and where the generated test cases are replayed in parallel. In the three first modes, generated tests are replayed after the termination of Klee in order to measure the reached coverage and to eliminate redundant tests in terms of coverage (with no guarantee of minimality).

We ran each mode on each program, with various coverage criteria: **DC**, **CC**, **MCC** and **WM**. For two programs having sufficiently relevant conditions (`check` and `gd`), we also used the **LIMIT** coverage criterion, which is **LIMIT-N** with $N = 0$. For each experiment, we limited the execution time of Klee to 60 s. Along with the total number of labels (#labels) added to express the selected criterion and the number of covered labels (#covered), we recorded the following measures. As reported by Klee, we have the number of LLVM instructions (#instr.) and the number of paths symbolically executed by Klee (#paths)[8]. We report the number of tests kept after replay (#tests, i.e. $|T'|$) and the total number of tests generated by Klee (gen, i.e. $|T|$). They may differ a bit even in the optimized mode[9]. Finally, we only report the total execution time of Klee as its execution time dominates every other computation of the whole front-end, including automated coverage label annotation by LAnnotate, and replays. We denote timeouts with 't.o.', and also denote with '—' the absence of any usable measure obtained from Klee after a timeout.

[6]We built Klee from its latest sources available at https://github.com/klee/klee on the the 25th June 2022 (commit `667ce0f1`), with STP 2.3.3 as default solver.

[7]More detailed results can be found the long version of the paper on https://doi.org/10.48550/arXiv.2211.14592.

[8]The number of paths is the sum of "complete paths" and "partially completed paths" reported by Klee.

[9]Tests reaching RTEs are filtered out as explained in Section 4.1. Moreover tests that only differ in the values assigned to non-deterministic inputs — the symbolic variables prefixed with `"nondet_"` — are unified.

RQ1. For simple criteria like **DC** or **CC**, the all-path coverage of Klee is sufficient to cover all labels. However, when aiming at complex criteria like **MCC**, **WM** or **LIMIT**, all-path coverage misses some behaviors most of the time. The need to help the exploration of the program with test objectives is prominent for these criteria. Comparing the results of the instrumented modes to the ignore mode, the label coverage is improved in 16 out of 27 experiments. With the optimized mode, the improvement spreads from 1,5% up to 600% (for `power` with **WM**), with a median of 14% and an average of 70%. This shows that label-guided test generation effectively leads to a better coverage of labels.

The coverage obtained in the ignore mode is on average 69% of the chosen criterion, while reaching 80% in the naive mode and 81% in the tight and optimized modes. In 10 out of 27 experiments, a coverage of more than 90% is achieved with the optimized mode. More accurate measures could be performed if we ruled out uncoverable labels (also called infeasible), since a coverage of 100% is not always possible. However, we have no immediate way to perform this analysis so far, and we leave the treatment of uncoverable labels as future work.

RQ2. Since each instrumentation mode adds paths to the original program, and Klee aims at covering all paths, the size of the generated test suite grows with the size of the path space. For example, on `tcas` with **WM**, as the number of explored paths grows from 44 (ignore), to 1484 (× 34, naive), to 2598 (× 59, tight), the number of generated tests grows from 18 (ignore), to 37 (× 2, naive), to 52 (× 2.9, tight). On some examples, the size of the test suites generated by the naive or the tight modes grows up to 7 (`checkutf8` or `tritype` with **WM**) and even 10 times (`fourballs` with **WM**).

On the contrary, with the optimized mode, the size of the generated test suite (i.e. $|T|$) is better controlled. For the examples mentioned above, the optimized mode generates 18 tests for `tcas` (× 1), 56 tests for `checkutf8` (× 2.4), 22 tests for `tritype` (× 1.6), 7 tests for `fourballs` (× 1.8). This grows up to × 4 for `power` with **WM**. Interestingly, the optimized mode does not lead to a larger test suite than the ignore mode in 17 cases out of 27, and it leads to a strictly smaller one in 8 cases, for a resulting coverage that is at least equal. The iterative deletion of labels that reduces the exploration to only yet uncovered labels proves to drastically limit the growth of the generated test suite.

As explained above, the generated test suite is filtered to eliminate tests reaching RTEs, as well as redundant tests that Klee may generate due to non-deterministic inputs. Therefore the number of tests actually kept in the resulting test suite (i.e. $|T'|$) is even smaller. The final test suite in the optimized mode is on average 0.98 times smaller than the one obtained with the ignore mode, and strictly smaller in 14 cases out of 27.

RQ3. The exploration time of Klee grows with the size of the path space, and as explained above, the path space grows with each instrumentation. This leads to an overhead

	$\mathscr{C}$ (#labels)	measure	ignore	naive	tight	optim
power (18 loc)	DC (4)	#covered	4	4	4	4
		#instr.	4692	8230	16 279	2192
		#paths	101	101	829	111
		#tests /gen	3 /3	3 /3	3 /4	2 /4
		time (s)	8.3	7.6	7.9	1.2
	MCC (4)	#covered	4	4	4	4
		#instr.	4692	8230	16 279	1783
		#paths	101	101	829	91
		#tests /gen	3 /3	3 /3	3 /4	3 /5
		time (s)	7.9	8.4	9.0	1.2
	WM (25)	#covered	3	22	21	21
		#instr.	10 284	—	58 957	69 195
		#paths	101	—	3358	3525
		#tests /gen	3 /3	9 /—	10 /26	7 /12
		time (s)	8.4	t.o.	t.o.	27.0
tritype (22 loc)	MCC (38)	#covered	27	38	38	38
		#instr.	473	2389	2711	2366
		#paths	14	27	172	152
		#tests /gen	14 /14	27 /27	26 /38	24 /24
		time (s)	0.8	1.7	1.7	1.7
	WM (101)	#covered	59	29	92	92
		#instr.	757	—	6955	5220
		#paths	14	—	363	272
		#tests /gen	14 /14	11 /—	33 /92	22 /22
		time (s)	0.5	t.o.	4.0	1.3
sel_sort (24 loc)	DC (8)	#covered	8	8	8	8
		#instr.	43 820	70 289	121 484	8413
		#paths	238	238	4200	322
		#tests /gen	6 /6	5 /5	4 /8	4 /6
		time (s)	9.6	9.3	15.0	1.2
	MCC (8)	#covered	8	8	8	8
		#instr.	43 820	70 289	121 484	8347
		#paths	238	238	4200	321
		#tests /gen	6 /6	5 /5	4 /8	4 /6
		time (s)	9.0	9.3	13.9	1.2
	WM (40)	#covered	27	31	31	31
		#instr.	71 784	388 367	287 329	382 183
		#paths	238	760	12 007	17 714
		#tests /gen	5 /5	9 /9	7 /31	6 /6
		time (s)	8.4	t.o.	t.o.	44.6
modulus (25 loc)	DC (8)	#covered	8	8	8	8
		#instr.	8042	13 717	22 654	1222
		#paths	159	166	1184	60
		#tests /gen	5 /6	5 /6	5 /10	3 /6
		time (s)	t.o.	t.o.	t.o.	1.2
	MCC (8)	#covered	8	8	8	8
		#instr.	8870	14 099	24 844	1131
		#paths	172	171	1294	54
		#tests /gen	5 /6	5 /6	5 /10	3 /6
		time (s)	t.o.	t.o.	t.o.	1.2
	WM (28)	#covered	7	14	7	7
		#instr.	12 286	9720	28 679	28 790
		#paths	156	75	1212	1220
		#tests /gen	5 /6	4 /6	5 /18	5 /14
		time (s)	t.o.	t.o.	t.o.	t.o.
fourballs (30 loc)	MCC (6)	#covered	6	6	6	6
		#instr.	262	319	384	376
		#paths	4	4	16	14
		#tests /gen	4 /4	4 /4	4 /6	4 /4
		time (s)	0.1	0.1	0.1	0.1
	WM (68)	#covered	39	43	43	43
		#instr.	392	2846	2355	1788
		#paths	4	42	103	79
		#tests /gen	4 /4	22 /22	5 /43	4 /7
		time (s)	0.1	t.o.	3.9	1.1
full_bad (65 loc)	MCC (34)	#covered	27	28	28	28
		#instr.	10 765	30 815	48 091	36 873
		#paths	207	235	2131	1641
		#tests /gen	12 /13	13 /15	14 /31	12 /13
		time (s)	14.0	24.9	26.5	17.9
	WM (86)	#covered	65	66	66	66
		#instr.	14 435	75 123	104 191	72 554
		#paths	207	357	3817	3408
		#tests /gen	12 /13	16 /18	13 /69	10 /13
		time (s)	12.8	41.2	47.8	23.8
checkutf8 (74 loc)	MCC (52)	#covered	42	42	42	42
		#instr.	10 503	29 169	62 549	50 170
		#paths	194	247	2936	2428
		#tests /gen	23 /23	24 /24	31 /42	23 /23
		time (s)	2.7	3.2	6.7	6.2
	WM (178)	#covered	80	146	143	143
		#instr.	17 801	305 984	169 042	126 977
		#paths	194	884	6884	6069
		#tests /gen	23 /23	37 /43	50 /159	44 /56
		time (s)	2.4	30.8	31.8	18.5
	LIMIT (25)	#covered	14	25	25	25
		#instr.	7963	966 752	33 773	17 803
		#paths	194	11 158	1561	853
		#tests /gen	23 /23	53 /53	25 /25	25 /26
		time (s)	2.0	49.3	6.5	3.7
replace (96 loc)	MCC (22)	#covered	17	17	17	17
		#instr.	21 220	41 908	69 542	60 141
		#paths	121	121	2333	2099
		#tests /gen	8 /8	8 /8	9 /17	7 /7
		time (s)	0.1	0.2	1.2	1.1
	WM (40)	#covered	23	26	26	26
		#instr.	27 124	208 937	151 659	123 351
		#paths	121	341	5135	5015
		#tests /gen	8 /8	10 /10	9 /26	3 /4
		time (s)	0.1	0.6	4.6	4.1
tcas (110 loc)	MCC (66)	#covered	51	53	53	53
		#instr.	8527	20 486	34 057	30 179
		#paths	44	68	1456	1363
		#tests /gen	23 /23	21 /21	17 /53	13 /13
		time (s)	0.4	0.7	1.9	2.2
	WM (87)	#covered	38	52	52	52
		#instr.	9977	861 437	56 147	47 106
		#paths	44	1484	2598	2027
		#tests /gen	18 /18	37 /37	29 /52	18 /18
		time (s)	0.6	22.7	8.3	3.6
get_tag (111 loc)	MCC (130)	#covered	63	63	63	63
		#instr.	231 544	1 286 607	460 391	449 894
		#paths	3714	3714	27 869	26 494
		#tests /gen	32 /32	27 /27	32 /67	27 /30
		time (s)	1.1	3.3	t.o.	t.o.
	WM (208)	#covered	129	135	135	135
		#instr.	241 482	1 620 399	1 067 978	924 940
		#paths	3714	4699	42 314	42 840
		#tests /gen	33 /33	36 /36	36 /135	22 /25
		time (s)	0.8	5.7	t.o.	t.o.
gd_full_bad (156 loc)	MCC (70)	#covered	57	59	59	59
		#instr.	541 633	794 240	1 225 078	1 019 620
		#paths	4201	4285	41 011	32 337
		#tests /gen	35 /35	43 /44	43 /61	30 /31
		time (s)	3.0	4.2	27.8	24.7
	WM (216)	#covered	139	159	158	158
		#instr.	633 653	5 728 717	1 328 045	1 158 999
		#paths	4201	19 401	45 734	46 290
		#tests /gen	32 /32	50 /53	40 /166	28 /31
		time (s)	3.8	44.5	t.o.	t.o.
	LIMIT (19)	#covered	6	16	16	16
		#instr.	492 819	1 609 082	611 393	560 088
		#paths	4201	14 435	11 192	7806
		#tests /gen	33 /33	46 /46	16 /16	16 /16
		time (s)	3.4	11.2	7.8	5.4

Table 1: Selected experimental results, where 't.o.' denotes a timeout (set to 60 s), and '—' denotes the absence of any usable measure obtained from Klee after a timeout.

of × 6.3 on average[10] for the naive mode (up to × 40 on `tcas` with **WM**), and of × 8.8 on average for the tight mode (up to × 50 on `fourballs` with **WM**). On the contrary, for the optimized mode, the overhead is better controlled, with an average overhead of × 4.6 and a maximum of × 30 for `replace` in **WM**.

On some examples, the optimized mode is even faster than the ignore mode: that is the case on programs with a lot of different paths but simple control-flow, like `power`, `selection_sort` and `modulus`. For example on `modulus`, the optimized mode ends after 1.2 s, while the ignore mode times out. On this kind of programs, for criteria like **DC**, **CC** or **MCC**, targeting labels drastically decreases the exploration time of Klee, for the same achieved coverage.

Interestingly, the overhead for the optimized mode drops down to a maximum × 2.2 when the chosen criterion can be fully satisfied. We noticed that Klee spends a lot of time on uncoverable labels. In fact, given a coverable label, either it is covered by the test specifically produced for it, or it is covered by executing another test, produced for another label. In the case of an uncoverable label, the latter cannot happen, so Klee needs to explore all the paths leading to this label, but finally fails to produce any test for it. This loss of time is particularly noticeable in the examples where all coverable labels are covered in less than 60 s (sometimes in the first ten seconds), but the timeout is reached trying to cover the uncoverable ones (e.g. `get_tag` with **MCC** and **WM**).

6 RELATED WORK

Test generation for labels. Support for labels was initially implemented inside the dynamic symbolic execution tool PathCrawler [2, 3]. This tool supports several control-flow coverage criteria from instruction to all-path coverage, and a bounded version of all-path named k-path coverage [31]. PathCrawler's depth-first search algorithm was extended to support labels in such a way that iterative label deletion is intimately interleaved with symbolic execution. The experiments show the efficiency of the approach with respect to the standard algorithm, allowing for a better label coverage (from 3% up to 39%) for only a slight overhead (median overhead of 37%, average overhead of 115%) [3].

Our support of labels for Klee follows a black-box approach, therefore the efficiency of the two tools cannot be directly compared. In particular, our instrumentation is totally independent of Klee's search algorithm, while the search algorithm of PathCrawler was significantly modified to support label coverage. Moreover, in our tool, the replay of produced test cases is done using the replay function provided by the API of Klee, which launches the executable from its very beginning. This induces an additional cost at each test execution, while this cost is minimized in the integration to PathCrawler. As the latter tool is not publicly available, we were not able to compare its efficiency to that of our implementation. Based on the available evaluation results [2], we achieved comparable results in terms of achieved coverage as well as execution time.

[10] We only compare experiments ending before the timeout.

The PathCrawler extension to labels has been used in industrial experimentations [4, 21]. The authors designed a unit test generation tool that combines several formal and non-formal test generation algorithms to target the MC/DC criterion. The C program is first annotated with coverage labels for this criterion, then several tools based on different techniques are called. After eliminating infeasible labels, they generate a test suite targetting the remaining labels. The tool uses a genetic algorithm, CBMC [6], and PathCrawler sequentially, to finally proceed to a test suite optimization phase. The authors achieve a coverage of 99% of MC/DC in about half an hour, on a case study of 82,000 lines of C code with integer data. This work gives interesting directions for improving efficiency and scalability. In particular, the detection of infeasible labels prior to test generation, and the collaboration between different tools, are the key to the scalability of the approach.

DSE enhanced with coverage criteria. Several work extended white-box testing techniques and tools to coverage criteria, for different programming languages. For example, in the Apex [16, 25] tool, which is an extension of Pex [30], coverage criteria like mutation or condition boundaries are added as constraints in the path conditions computed by the dynamic symbolic execution. Another extension of Pex for mutation testing, PexMutator [32], instruments the program under test with weak-mutant-killing constraints. Each of these constraints is wrapped as a conditional statement, and Pex is then called on the resulting program. This approach compares to what we call naive instrumentation in Section 4.1. Another comparable approach is adopted for white-box testing for Java by Papadakis and Nicos [27], where weak mutations are encoded as new branching conditions in the source code, of the form *original expression ≠ mutated expression*. None of these works tackle the path explosion problem caused by the addition of branching conditions.

Assertion and RTE checking. Besides traditional coverage criteria, users often want to test given properties of their programs, like the absence of certain runtime errors and the satisfaction of assertions. This is the aim of assertion-based testing [10, 14, 19], where the analysis of the program is reduced to the scope of the targeted property. Godefroid et al. [14], in particular, uses dynamic symbolic execution to compute the set of all the paths reaching the targeted property: if none of the resulting path conditions is satisfiable, the unreachability of the property is proven; otherwise, a counterexample gives a test case.

In a similar way, CBMC [11] performs bounded model-checking on a C source code, in order to check for reachability properties (hand-written or known runtime errors). The tool is also able to perform test case generation along different coverage criteria, automatically encoded as reachability properties in the code [1]. This instrumentation of the code with extra assertions to encode criteria makes it very close to our annotation with labels. However, since model-checking does not explore program paths one by one, no extra instrumentation is needed and the BMC algorithm is very efficient when treating these additional assertions.

7 CONCLUSION AND FUTURE WORK

A larger support of various coverage criteria in test generation tools remains a challenging research topic. Coverage labels offer a unified framework for specifying coverage criteria in a generic way. However, labels are still not supported in popular test generation tools. Inspired by a previous theoretical proposal for an efficient test generation technique for labels by Bardin et al. [2], in this work we show how to integrate a dedicated support for labels into Klee [6], a popular open-source test generation tool based on dynamic symbolic execution. We perform a lightweight black-box integration, which does not need to modify the underlying test generation strategy and can therefore directly benefit of various strategies and optimizations of the tool. We expect that this work will facilitate and guide a lightweight integration of this technique into other similar tools. Experiments with our version of the tool on several benchmarks confirm the benefits of the proposed approach. In particular, our approach efficiently achieves basic criteria while generating fewer and more targeted test cases than when Klee is used directly. On more advanced criteria like multiple conditions, weak mutations, and condition limits, our approach is capable of efficiently achieving high coverage with a reasonable overhead.

Future work includes a large industrial evaluation of the proposed approach on real-life code. Detecting infeasible objectives before running test generation is another promising perspective, since it can avoid the waste of time and effort of test generation tools trying to cover test objectives that cannot be covered. It can rely on tools like CBMC [11] and the LUncov module of LTest [3]. Another future work direction is a further extension of the proposed approach to other criteria not expressible with labels [23, 24]. Finally, integration of the proposed approach into other tools, based on dynamic symbolic execution or other test generation techniques like fuzzing, can be an interesting extension for these tools.

Acknowledgement. Part of this work was funded by the French Defense Innovation Agency via RAPID project TAGAda.

REFERENCES

[1] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. 2010. Using Bounded Model Checking for Coverage Analysis of Safety-Critical Software in an Industrial Setting. *J. Autom. Reason.* 45, 4 (2010), 397–414.

[2] Sébastien Bardin, Nikolai Kosmatov, and François Cheynier. 2014. Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria. In *ICST*. 173–182.

[3] Sébastien Bardin, Nikolai Kosmatov, Michaël Marcozzi, and Mickaël Delahaye. 2021. Specify and Measure, Cover and Reveal: A Unified Framework for Automated Test Generation. *Sci. Comput. Program.* 207 (2021), 102641.

[4] Sébastien Bardin, Nikolai Kosmatov, Bruno Marre, David Mentré, and Nicky Williams. 2018. Test Case Generation with PathCrawler/LTest: How to Automate an Industrial Testing Process. In *ISOLA (LNCS)*, Vol. 11247. 104–120.

[5] Dirk Beyer. 2022. Advances in Automatic Software Testing: Test-Comp 2022. In *FASE (LNCS)*, Vol. 13241. 321–335.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. 209–224.

[7] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *CCS*. 322–335.

[8] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *ICSE*. 1066–1071.

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

[10] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. 2012. Program Slicing Enhances a Verification Technique Combining Static and Dynamic Analysis. In *SAC*. 1284–1291.

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS (LNCS)*, Vol. 2988. 168–176.

[12] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empir. Softw. Eng. J.* 10, 4 (2005), 405–435.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *OSDI*. 213–223.

[14] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Active Property Checking. In *EMSOFT*. 207–216.

[15] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*.

[16] Konrad Jamrozik, Gordon Fraser, Nikolai Tillman, and Jonathan de Halleux. 2013. Generating Test Suites with Augmented Dynamic Symbolic Execution. In *TAP (LNCS)*, Vol. 7942. 152–167.

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

[18] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A Software Analysis Perspective. *Form. Asp. Comput.* 27 (2015), 573–609. Issue 3.

[19] Bogdan Korel and Ali M. Al-Yami. 1996. Assertion-Oriented Automated Test Data Generation. In *ICSE*. 71–80.

[20] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. 2007. A Buffer Overflow Benchmark for Software Model Checkers. In *ASE*. 389–392.

[21] Éric Lavillonnière, David Mentré, and Denis Cousineau. 2019. Fast, Automatic, and Nearly Complete Structural Unit-Test Generation Combining Genetic Algorithms and Formal Methods. In *TAP (LNCS)*, Vol. 11823. 55–63.

[22] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. 1997. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *MICRO*. 330–335.

[23] Michaël Marcozzi, Michaël Delahaye, Sébastien Bardin, Nikolai Kosmatov, and Virgile Prevosto. 2017. Generic and Effective Specification of Structural Test Objectives. In *ICST*. 436–441.

[24] Thibault Martin, Nikolai Kosmatov, Virgile Prevosto, and Matthieu Lemerre. 2020. Detection of Polluting Test Objectives for Dataflow Criteria. In *iFM*. 337–345.

[25] Rahul Pandita, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2010. Guided Test Generation for Coverage Criteria. In *ICSM*. 1–10.

[26] Mike Papadakis, Nicos Malevris, and Maria Kallia. 2010. Towards automating the generation of mutation tests. In *AST*. 111–118.

[27] Mike Papadakis and Malevris Nicos. 2011. Automatically Performing Weak Mutation with the Aid of Symbolic Execution, Concolic Testing and Search-Based Testing. *Software Qual. J.* 19, 4 (2011), 691–723.

[28] Corina S Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *ASE*. 179–180.

[29] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *FSE*. 263–272.

[30] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex–White Box Test Generation for .NET. In *TAP (LNCS)*, Vol. 4966. 134–153.

[31] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. 2005. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In *EDCC (LNCS)*, Vol. 3463. 281–292.

[32] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, and Hong Mei. 2010. Test Generation via Dynamic Symbolic Execution for Mutation Testing. In *ICSM*. 1–10.