Registered Report: Fine-Grained Coverage-Based Fuzzing

Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin
Université Paris-Saclay, CEA, List
Palaiseau, Paris Metropolitan Area, France
first.last@cea.fr

***Abstract*—Fuzzing is an effective software testing method that discovers bugs by feeding target applications with (usually a massive amount of) automatically generated inputs. Many state-of-art fuzzers use branch coverage as a feedback metric to guide the fuzzing process. The fuzzer retains inputs for further mutation only if branch coverage is increased. However, branch coverage only provides a shallow sampling of program behaviours and hence may discard inputs that might be interesting to mutate. This work aims at taking advantage of the large body of research over defining finer-grained code coverage metrics (such as mutation coverage) and use these metrics as better proxies to select interesting inputs for mutation. We propose to make coverage-based fuzzers support most fine-grained coverage metrics out of the box (i.e., without changing fuzzer internals). We achieve this by making the test objectives defined by these metrics (such as mutants to kill) explicit as new branches in the target program. Fuzzing such a modified target is then equivalent to fuzzing the original target, but the fuzzer will also retain inputs covering the additional metrics objectives for mutation. We propose a preliminary evaluation of this novel idea using two state-of-art fuzzers, namely AFL++(3.14c) and QSYM with AFL(2.52b), on the four standard LAVA-M benchmarks. Significantly positive results are obtained on one benchmark and marginally negative ones on the three others. We discuss directions towards a strong and complete evaluation of the proposed approach and call for early feedback from the fuzzing community.**

I. Introduction

Context. Fuzzing [1] refers to a process of repeatedly running a Program Under Test (PUT) with automatically generated inputs to trigger faults [2]. The motive is to detect bugs as early as possible, before they cause failures or get exploited as vulnerabilities in production [3]. Fuzzing has gained much attention in recent years; researchers and practitioners have notably proposed various methods to improve the input generation process. Many state-of-the-art fuzzers use a mutation- and coverage-based approach to generate new inputs from the ones generated before (Section III). As inputs are being generated, those that cover yet uncovered parts of the PUT are saved and randomly mutated (i.e. slighlty modified) to generate novel inputs, possibly covering even more new parts of the PUT. American Fuzzy Lop (AFL/AFL++) [4], [5] is one the most used and forked tools relying on such an approach, while many other tools have tried to improve this approach, notably by combining it with program analyses like symbolic execution (e.g. Qsym [6], Driller [7]) or taint analysis (e.g. Vuzzer [8]).

Problem. In order to select which of the generated inputs will be saved for subsequent mutation, current fuzzers run the PUT with these inputs and measure some form of branch coverage. Inputs that cover branches of the PUT that have not been previously covered during the fuzzer run are then typically selected. Yet, branch coverage is a shallow metric to evaluate how well the possible behaviours of a program are exercised, so that a cornerstone of software testing research has been the definition and evaluation of finer-grained coverage metrics [9] (Section III), such as multiple condition coverage or mutation testing. To the best of our knowledge (Section VIII), there has been no effort so far to support fine-grained coverage metrics within state-of-the-art fuzzers, and the ability of such an integration to improve fuzzers remains unknown. Yet, by making fuzzers more sensitive in retaining and mutating inputs that triggers new behaviours of the program, one may hope that it could make PUT exploration and bug detection more efficient.

Goal and challenge. In this work, we aim at (1) providing support for fine-grained coverage metrics as a means to select inputs for mutation in a wide range of state-of-the-art coverage-based fuzzers and (2) evaluating the impact of using such metrics in addition to branch coverage over the efficiency of fuzzers to exercise the PUT and find bugs in it. A significant challenge to overcome to reach these goals is harnessing the variety of fuzzers to augment and of fine-grained metrics to support, in order to obtain significant results. In particular, we do not want to dig into the internals of every fuzzer and find a way to extend them with support for every additional criterion.

Proposal. We propose to make state-of-the-art coverage-based fuzzers support most fine-grained coverage metrics out of the box (i.e. without changing their internals), by relying on a dedicated transformation of the code of the PUT (Sections II and IV). We take advantage of the fact that the coverage objectives defined by most fine-grained coverage metrics (like conditions to activate or mutants to kill) can be made explicit in the code of the PUT in a generic way and without modifying its semantics [10]. More precisely, given a PUT and a fine-grained coverage metric, we instrument the code of the PUT with new branches corresponding to the objectives from the metric. Covering one of these branches is then equivalent to covering the corresponding objective from the metric. Fuzzing such a transformed PUT with a coverage-based fuzzer (relying

International Fuzzing Workshop (FUZZING) 2022
24 April 2022, San Diego, CA, USA
ISBN 1-891562-77-0
https://dx.doi.org/10.14722/fuzzing.2022.23xxx
www.ndss-symposium.org

on branch coverage) is then equivalent to fuzzing the original PUT, but with the fuzzer also saving for mutation the inputs that cover additional objectives from the metric.

Preliminary evaluation. We chose to evaluate our approach using the Multiple Condition Coverage and Weak Mutation Coverage metrics. These metrics are notoriously stronger than Branch Coverage. Multiple Condition Coverage can help fuzzing performance by systematically retaining inputs that trigger subtle variations within the program's control-flow logic, not captured by branch coverage. Weak Mutation Coverage can help fuzzing performance by systematically retaining inputs that would make any common programming mistake possibly present in the program corrupt the program state.

We developed a tool (based on Clang) to automatically instrument C code with the Multiple Condition Coverage and Mutation Coverage metrics objectives (Section V). We use this tool to transform the four programs from Lava-M [11], a standard and basic fuzzer benchmark. We run the AFL++ and Qsym fuzzers five times for 24 hours over the original Lava-M programs and their transformed versions (Section VI).

With one of the four Lava-M programs (*who*), we observe significantly better fuzzing results after transforming the program, with, on average, 100 more bugs being discovered in total and bugs being uncovered faster during the fuzzing process. With the three other Lava-M programs, the results indicate marginally worse results after transforming the program.

Discussion and future work. While the positive results obtained with the *who* program encourage us to explore even more the potential of our approach, we discuss in the paper the main directions to achieve a strong and complete evaluation of it in the future (Section VII). These directions involve fast pruning of infeasible test objectives before fuzzing, extensive evaluation with more diverse benchmarks, discriminating the impact of specific coverage criteria, using more standard fuzzer evaluation metrics, as well as measuring impact over fuzzer throughput.

II. Motivating example

We illustrate now with a simple example how our approach can make a state-of-the-art fuzzer support a fine-grained coverage metric out-of-the-box, by transforming the code of the program under test. We also exemplify how this approach could end up making coverage-based fuzzers more efficient at finding bugs, by making them more sensitive in retaining and mutating inputs that trigger different PUT behaviours.

Our example PUT is the C program presented in Listing 1. It is basically a C function checking if an appliance is running outside its allowed temperature range and taking corrective actions if so. We suppose that the implementation of these corrective actions is buggy, but that the bug only triggers a program crash (enabling a fuzzer to detect it) when the temperature is negative and other rare conditions are met (requiring the fuzzer to generate many inputs with `current_temp` less than -50 to actually trigger it).

The Condition Coverage criterion is a fine-grained code coverage metric requiring both truth values of any atomic condition appearing at the decision points of the program to

```
1  void check_temperature_ok(int current_temp,char *data[] ){
2  if(current_temp>50 || current_temp<-50)
3      {
4      // Deal with appliance running outside
5      // the allowed temperature range
6
7      ... // Buggy code, but the crash only
8          // triggers scarcely and
9          // when current_temp is negative
10
11     }
12 }
```

Listing 1: A buggy program checking if an appliance is running outside its allowed temperature range and taking corrective actions if so.

```
1  void check_temperature_ok(int current_temp,char *data[]){
2  if(current_temp>50) {}
3  if(current_temp<=50) {}
4  if(current_temp<-50) {/* Mutating seed entering here
5                           helps triggering crash */ }
6  if(current_temp>=-50) {}
7  if(current_temp>50 || current_temp<-50)
8      {
9      // Deal with appliance running outside
10     // the allowed temperature range
11
12     ... // Buggy code, but the crash only
13         // triggers scarcely and
14         // when current_temp is negative
15
16     }
17 }
```

Listing 2: Same program as Listing 1, but with instrumentation for fine-grained fuzzing with the CC criterion.

be activated. In our example program, this means requiring to generate inputs where the value of `current_temp` is respectively greater than 50, not greater than 50, less than -50 and not less than -50. Our approach makes these four coverage objectives explicit in the code of the PUT, by adding four conditional statements corresponding respectively to each of them, as illustrated in Listing 2. The transformed program can then be fuzzed using an off-the-shelf coverage-based fuzzer relying on branch coverage. Every time this coverage-based fuzzer will produce an input entering one of these new conditionals (and thus satisfying an additional Condition Coverage objective), it will save it for mutation.

It should be remarked that the third added conditional explicitly forces the fuzzer to maintain and mutate an input where `current_temp` is less than -50 as soon as it generates one. This will increase the chance for the fuzzer to trigger a crash revealing the bug, making bug detection faster in average.

III. Background

A. Coverage-based fuzzing

Coverage-based fuzzing uses a feedback mechanism to improve the efficiency and effectiveness of the input generation process. Figure 1 shows the general working principle of a coverage-based fuzzer. It starts with an initial set of user-provided input seeds; if unavailable, the fuzzer will construct one [12][13] by itself. Then, the fuzzer mutates these initial seeds and executes the program under test with the resulting inputs. If the execution exercises new control-flow edges

(a.k.a. code branches), the input is considered as interesting and kept as a seed for further mutation; otherwise, it is discarded. Usually, the coverage information is monitored

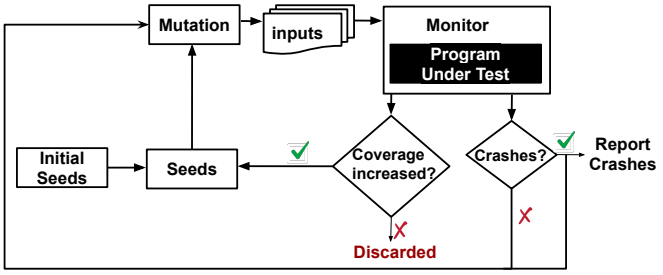


Fig. 1: General coverage-based fuzzing process

using lightweight program instrumentation and hence does not hinder the program's execution speed. This simple technique has proved to be very effective in finding bugs in real-world applications. The popular implementation of coverage-based fuzzer are AFL [4] and AFL++ [5]; its variants are notably Steelix [2], Driller [7], libFuzzer [14], AFLFast [15] and QSYM [6].

B. Code coverage criteria

Code coverage metrics, commonly referred to as code coverage criteria, are a cornerstone of software testing research. They have been studied for decades in the literature [16] [17] [18], and are notably used to evaluate the effectiveness of test suites to exercise a piece of software. This can involve properly testing for functional correctness, security, reliability, or performance. We list hereafter a few standard classes of coverage criteria and their most common criteria.

Control-flow and call graph criteria.

- *Statement Coverage (SC):* requires a test suite to reach each statement of the program under test
- *Decision Coverage (DC):* requires a test suite to activate both the true and false path of each decision point in the program under test. This is equivalent to covering all the edges in the control-flow graph of the program. The test objectives defined by DC (control-flow edges) are thus at the heart of the input generation heuristics used by coverage-based fuzzers.
- *Function Coverage (FC):* requires a test suite to reach all function entry-points.

Logic expressions criteria.

- *Condition Coverage (CC):* requires a test suite to activate both true and false values for each of the *atomic conditions* in any program decision point. Here, *atomic conditions* refers to logical expressions that cannot be divided into other simpler expressions.
- *Decision Condition Coverage (DCC):* requires a test suite to satisfy both DC and CC.
- *Multiple Condition Coverage (MCC):* requires a test suite to activate all the combinations of truth values of all atomic conditions at each decision point in the program.

Mutation criteria. Mutation criteria are derived from the research efforts in mutation testing [19]. Test objectives consist here of mutants, i.e. slight variants of the program under test. The goal of mutation testing is to help improve the quality of test suites by checking whether or not they are able to elicit the common programming mistakes that could be present in the code, i.e. differentiate the PUT from its mutants. We say that an input from a test suite kills a mutant if it triggers an observable difference between the PUT and the mutant. If this this difference is observable from outside of the code, as the PUT and the mutant generate different outputs, we say that the input *strongly* kills the mutant. If the difference is only observable in the internal states of the PUT and mutant around the mutation points, but do not propagate to their outputs, we say we say that the input *weakly* kills the mutant. From this, we can define the strong and weak mutation coverage criteria:

- *Strong Mutation coverage (SM):* requires a test suite to kill strongly all the created mutants of the program.
- *Weak Mutation coverage (WM):* requires a test suite to kill weakly all the created mutants of the program.

In order to create a significant set of mutants for a given PUT, one should use standard mutant creation operators, like Absolute Value Insertion *(ABS)*, Arithmetic Operator Replacement *(AOR)*, Conditional Operator Replacement *(COR)* and Relational Operator Replacement *(ROR)* for programs written in simple imperative style.

About criteria diversity. In addition to the criteria presented hereabove, the scientific literature describes an even wider range of diverse code coverage criteria, enabling to focus over different aspects of program behaviour. This variety of criteria also offers a variety of different balances between test thoroughness and speed. Lightweight criteria (like statement or decision coverage) favor small but shallow test suites, while heavyweight criteria (like multiple condition or mutation coverage) favor thorough but large (and thus slower) test suites.

C. Making test objectives explicit with labels

Bardin et al. [10] [20] [21] have proposed a generic mechanism for specifying the test objectives from many coverage criteria explicitly within the PUT code. This mechanism relies on *labels*, i.e., predicates attached to program locations. A label is covered by an input if executing the program with this input enables reaching the location and satisfying the predicate. Figures 2 and 3 illustrate how the test objectives from the condition, multiple condition and weak mutation coverage criteria can be made explicit by a corresponding label. Covering the label is then equivalent to covering its corresponding test objective. While a significant set of coverage criteria can be handled in this way, the expressive power of labels alone is not sufficient to make the test objectives from all criteria explicit (e.g. strong mutation). Marcozzi et al. [22] discuss this issue and extend labels into hyperlabels to handle a wider set of coverage criteria.

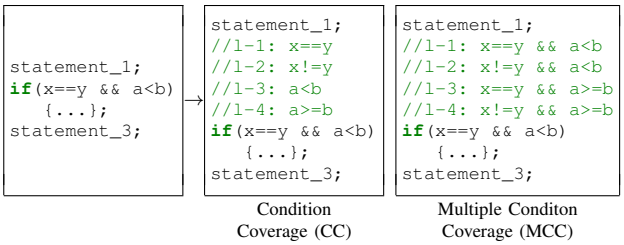


Fig. 2: Encoding CC and MCC test objectives with labels [10].

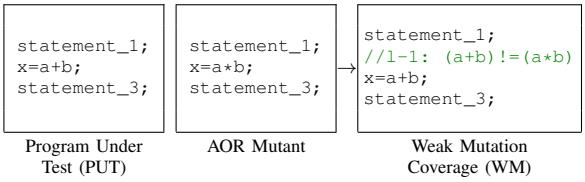


Fig. 3: Encoding WM test objectives with labels (considering a single mutant created with the AOR operator).

IV. Fine-grained coverage-based fuzzing

A. General principle

Given a program P and a label-encodable code coverage criterion C, our approach transforms the original program P into a semantically equivalent program P_{lannot}, so that fuzzing P_{lannot} with a coverage-based fuzzer is the same as fuzzing P and keeping the inputs that increase coverage w.r.t. C as additional seeds for subsequent mutations.

In practice, transforming P into P_{lannot} works as follows. For each label l corresponding to one of the test objectives required by the coverage criterion C for P (e.g. truth values to activate, mutants to kill), we add an empty conditional statement if at the same location in P as l and whose entry condition is l's predicate. This transformation process is illustrated on a simple code snippet at Figure 4. When fuzzing P_{lannot}, the fuzzer will save as a seed for mutation any input that covers a previously uncovered code branch. If this branch is the one satisfying the entry condition of if, the fuzzer will basically save as a seed an input covering l and its corresponding objective from C.

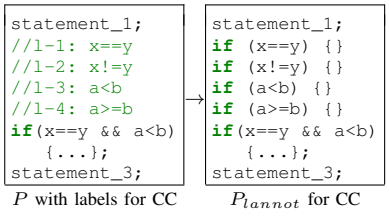


Fig. 4: Transforming a program for fine-grained coverage-based fuzzing with CC criterion [10].

B. Handling of expressions with side-effects

Applying our code transformation approach to programs involving expressions with side-effects may alter the semantics of these programs, in case such expressions end up being part of the considered label predicates. This is illustrated in Figure 5, where the transformed program would print "aabbab" instead of just "ab", like in the original program.

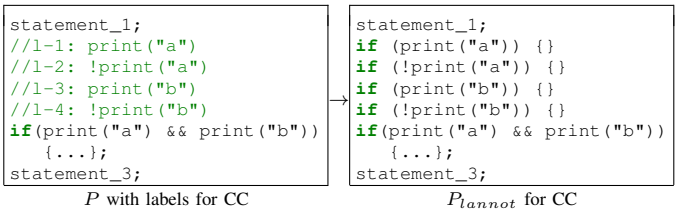


Fig. 5: Naive transformation of a program containing expressions with side-effects.

To preserve the semantics of P under transformation into P_{lannot}, we first transform P into a normalised program $P_{normalised}$. $P_{normalised}$ is obtained from P by extracting the side-effects from all the expressions involved into a label predicate, without modifying the semantics of P. This normalization process is illustrated in Figure 6 for side-effects appearing in the atomic conditions of decision points in the program (such conditions are involved in label predicates for many coverage criteria). Case (a) details the simple situation where the atomic condition with side-effects can be extracted into a new temporary variable defined just before the decision point. Such an unconditional extraction is not possible in case the evaluation of the atomic condition can be short-circuited because of a lazy boolean operator. Case (b) and (c) detail the conditional extraction performed in this situation, respectively for a lazy AND and OR operator.

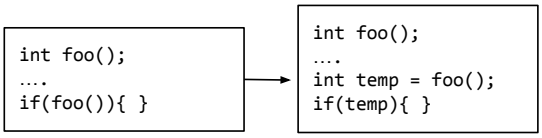
C. Complete workflow

Figure 7 summarises the complete workflow of our fine-grained coverage-based fuzzing approach. After extracting side-effects from expressions involved in label predicates and adding conditionals corresponding to these labels making the criterion explicit, the transformed PUT is fuzzed using a classical (branch coverage-based) fuzzer. The transformation guarantees that this is equivalent to fuzzing the original PUT, but with the fuzzer also saving for mutation the inputs that cover additional objectives from the criterion.

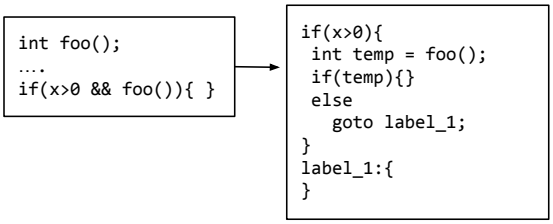
V. Implementation

We have implemented our program transformation approach for C programs as passes in the Clang open-source compiler infrastructure [23]. Our implementation involves recursively traversing the Abstract Syntax Tree (AST) of our target program using recursive visitors provided by the *Clang API*. Our program normalisation pass transforms boolean conditions at program decision points if they contain side-effects, like function calls. Our program annotation pass inserts conditional statements for the MCC and WM criteria, using the *Rewriter* class provided by the *Clang API*.

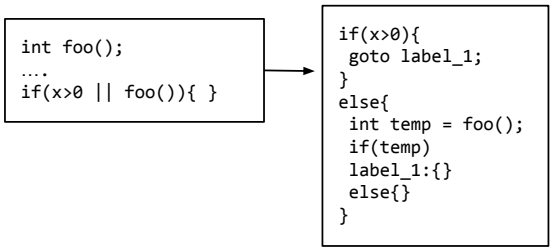
We have also carefully examined by hand the binary code produced after transforming a sample program with our passes, instrumenting the transformed code with the fuzzer harness and compiling the result with gcc, in order to make sure that our transformation was not tampered by the harnessing or compilation processes.



(a) Side-effect in an atomic condition at a decision point



(b) Side-effect in the second atomic condition of a lazy boolean operator (AND case)



(c) Side-effect in the second atomic condition of a lazy boolean operator (OR case)

Fig. 6: Extracting side-effects from atomic conditions at program decision points.

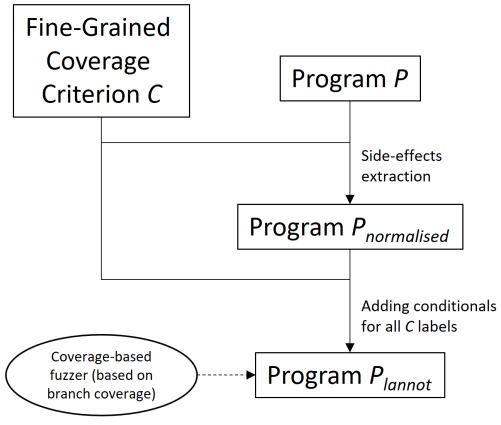


Fig. 7: Fine-grained coverage-based fuzzing workflow

VI. Preliminary Experimental Evaluation

As a preliminary evaluation, we use our code transformation tool to add conditional statements for MCC and WM objectives to the four programs of the standard LAVA-M benchmark suite [11]. Then, we compare the performance of the *afl++3.14c* and *QSYM* coverage-based fuzzers on these programs with and without our conditional statements added (in terms of bug discovery, code coverage and time to bug). We consider the following research questions:

- *Research Question 1 (RQ1)* Is our code transformation tool effective and useful? (a) Is it easy to use and does out-of-the-box integration with existing fuzzers work well? (b) Can it scale to real-world applications?
- *Research Question 2 (RQ2)* Does our fine-grained approach allow to improve over the baseline state-of-art fuzzers?

A. Experimental setup

We set up two different settings of experiments for each fuzzer. One fuzzing the original programs (baseline) and other fuzzing these programs with the *label-derived conditional statements added* (our approach). Each of the four settings had the same time budget of 24 hours. We used an Intel Skylake CPU, with 192GB memory RAM and 72 logical cores running at 2.6GHz. To mitigate the impact of randomness, we run five trials for AFL++ and QSYM in both settings. To evaluate performance, we measure the number of unique bugs found along time (using the built-in LAVA-M capabilities) and the MCC and WM objectives covered along time (we monitor the inputs generated by the fuzzers and re-run the program on these inputs with label instrumentation to obtain the label coverage measurement).

B. RQ1: effectiveness and scalability

We consider the four programs from the LAVA-M benchmark suite [11] (built on top of coreutil-8.14). LAVA-M is designed to evaluate the effectiveness of fuzzers by injecting synthetic bugs in different execution paths. Our automated tool successfully generated five test objective (i.e. labels) types, i.e., MCC and WM with ABS, AOR, COR, and ROR operators. Table I shows the number of labels generated for the LAVA-M benchmark suite. The time taken to instrument each program is roughly five seconds. Once we have the transformed source code, we can now run any off-the-shelf fuzzer for fuzzing it. Our approach was thus tested as practical, effective, easy to used for the small size programs of the LAVA-M benchmark. Table I, column two shows the Line of Code (LOC) for each applications of the LAVA-M ranging from 255 to 663 LOC.

TABLE I: Number of labels on LAVA-M benchmark suite. LOC: Line of Code, MCC: Multiple condition coverage, ABS: Absolute value insertion, AOR: Arithmetic operator replacement, COR: Conditional Operator Replacement, ROR: Relational Operator Replacement

Application	LOC	MCC	ABS	AOR	COR	ROR	Total
uniq	494	204	61	7	18	45	335
base64	255	26	56	7	6	51	146
md5sum	663	125	113	20	24	79	361
who	622	170	180	15	30	19	414

C. RQ2: comparison with state-of-the-art fuzzers

We evaluate our technique using two state-of-the-art fuzzers, i.e., *afl++3.14c* and *QSYM*. For each program, we ran five trials for 24hrs which accounted for 1920 CPU hours. We compare our approach as follows:

1) Bugs discovery: Table II shows the number of bugs found by the two fuzzers in both settings. Columns 1, 2, and 3 show the program's name, arguments, and the listed bugs by LAVA-M Authors, respectively.

Regarding the AFL results, columns 4 and 5 are the number of bugs found by *alf++3.14c* and *afl++3.14c+lannot* respectively. Our evaluation shows a single difference: *afl++3.14c+lannot* finds two bugs in *base64*, whereas *afl++3.14c* found no bugs.

Regarding the QSYM results, columns 6 and 7 show the number of bugs found by *QSYM* and *QSYM+lannot*, respectively. Our evaluation demonstrate a significant improvement with our approach in the *who* program, as 101 additional bugs were triggered on average. However, the baseline is still better in the maximum number of bugs taken from all trials. For the three other programs, the baseline is similar or marginally better than our approach (0-3 more bugs found on average).

2) Coverage: Our evaluation is on the number of *labels* (i.e. MCC or WM objectives made explicit) covered by the two fuzzers in both settings. Figure 1 to 4 shows the cumulative label coverage (in %) on the LAVA-M benchmark. Our approach shows degrading in coverage in *base64* and *md5sum*, however our approach shows improvement *uniq* and *who* by 0.3% and 1.23% respectively. Table III shows the absolute number of recovered labels by fuzzers. For each application, we show the minimum, average and maximum number of labels covered by each fuzzers. We observed in three applications i.e., *uniq*, *base64*, and *md5sum* the number of labels are closely the same except in *md5sum* where the minimum number of labels are 6 less as compared to the baseline in AFL++. Our approach shows effectiveness in *who* application with *+4*, *+5* and *+4* in minimum, average and maximum of labels triggered by QSYM respectively.

3) Time to bugs: Figure 9 shows the average number of new unique bugs found by *QSYM* and *QSYM+lannot* along fuzzing time. Here again, we observe a similar pattern: encouraging results for the *who* program and similar or marginally worse results for the three other programs.

VII. Discussion and future work

While the positive results obtained with the *who* program encourage us to explore even more the potential of our fine-grained coverage-based fuzzing approach, a lot of work remains to be done to provide a strong and complete evaluation of it. We discuss the main directions to achieve such a better evaluation in the next paragraphs.

First, Figure 8 reveals that, even after 24 hours of fuzzing, more than 70% of the MCC and WM test objectives remain uncovered, whatever the PUT or fuzzer used (as label coverage tops at 27.5% for AFL++ with *md5sum*). As discussed in [24], a significant proportion of labels (test objectives) can be infeasible (i.e. no input can cover them). Infeasible labels may be the possible reason of such a low level of coverage. In addition, many infeasible labels means that our program transformation will add a lot of conditional statements with an unsatisfiable entry condition. This may penalize the performance of fuzzers on our transformed PUT, as it adds a lot of useless dead code, and in the case of QSYM, saturate the symbolic execution module with a lot of infeasible constraints. It looks thus essential to prune out such infeasible labels. Bardin et al. [10] [24] [25] proposed a sound and fast approach to prove that many labels are infeasible using static analysis, while Papadakis et al. [26] proposed Trivial Compiler Equivalence (TCE) to detect equivalent mutants (in our case, infeasible labels) by exploiting off-the-shelf compiler optimisations. We plan to leverage one of these two approaches to prune out infeasible labels and report on the impact over fuzzing performance.

Second, it remains unclear why the results are positive on the *who* program, but marginally negative on the three other ones. As a way towards solving this issue, we plan to gather additional results over a more diverse set of programs, to see if a more consistent trend emerges. More precisely and also in order to demonstrate the scalability of our approach, we plan to test it on real-world applications praised by the fuzzing community as ground-truth benchmarks, such as Magma [27].

Third, our current experiments use as a coverage metric the combined coverage of all types of labels (MCC and WM objectives for all operators), taken as a whole. It would be interesting to investigate considering each type of labels separately in the future. This will give us an idea and quantitative suggestion as to which coverage criteria really help the fuzzing.

Fourth, our evaluation considers label coverage, i.e. our own internal metric, to measure the performance of the two fuzzer in both settings. It seems important to also evaluate our approach using more standard metrics, like AFL++'s edge coverage, as suggested by the fuzzing community [28].

Fifth, our method introduces additional branches, which makes the fuzzer retain more inputs as seeds, impacting the fuzzing throughput (generated inputs per seconds). We plan to study the impact of labels on throughput in the future.

VIII. Related work

A. Improving input selection with additional fuzzing objectives

Aschermann et al. [29] propose IJON: a human-in-the-loop technique that gives feedback to the fuzzer. The user first identifies hard-to-cover code and then annotates it with special primitives to capture the associated program states. The annotation process requires domain knowledge of the target program and much manual work. Additionally, modifying the fuzzer itself is needed for it to capture the program states. In contrast, our work aims at using various code coverage criteria from the software testing literature to guide the fuzzer; in addition, our code annotation is done automatically and there is no need for any modification of existing fuzzers.

Wang et al. [30] study the performance impact of implementing different variants of the branch-coverage metric within the fuzzer. They also provide a theoretical concept of metric sensitivity that can be used to compare different coverage metrics. The study shows that no branch coverage variant surpasses the others. For instance, a more sensitive variant may choose more inputs as seeds, which results in the fuzzer not having enough time to schedule or adequately mutate all of the seeds; thus, reducing fuzzer throughput. On the other hand, less sensitive variants may choose fewer inputs as seeds; hence, potentially missing some intriguing ones. To address this problem, Wang et al [31] proposed AFL-HIER: a

TABLE II: Bugs found on the LAVA-M benchmark by fuzzers. Here, min, avg, and max are the minimum, average and maximum number of bugs found by fuzzers, respectively. Note that fuzzers sometimes find more bugs than those listed by LAVA authors.

Program	Argument	Listed bugs	Number of bugs							
			AFL++	AFL++ +*lannot*	QSYM			QSYM+*lannot*		
			max	max	min	avg	max	min	avg	max
uniq		28	1	1	15	19	29	11 (-4)	19	27 (-2)
base64	-d	44	0	**2**	43	45	47	41 (-2)	44 (-1)	47
md5sum	-c	57	0	0	40	44	60	40	42 (-2)	50 (-10)
who		2136	1	1	1599	1786	2110	1734 (-135)	1887 (+101)	2020 (-90)

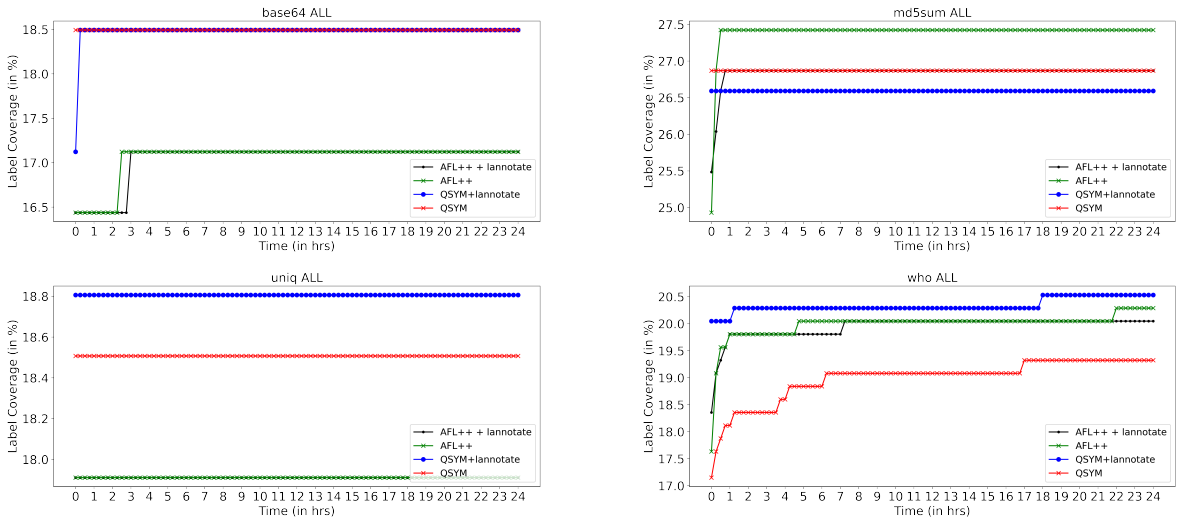


Fig. 8: Cumulative Label coverage (in %) on LAVA-M by AFL++, AFL++ + *lannotate*, QSYM, and QSYM + *lannotate* in 24 hours

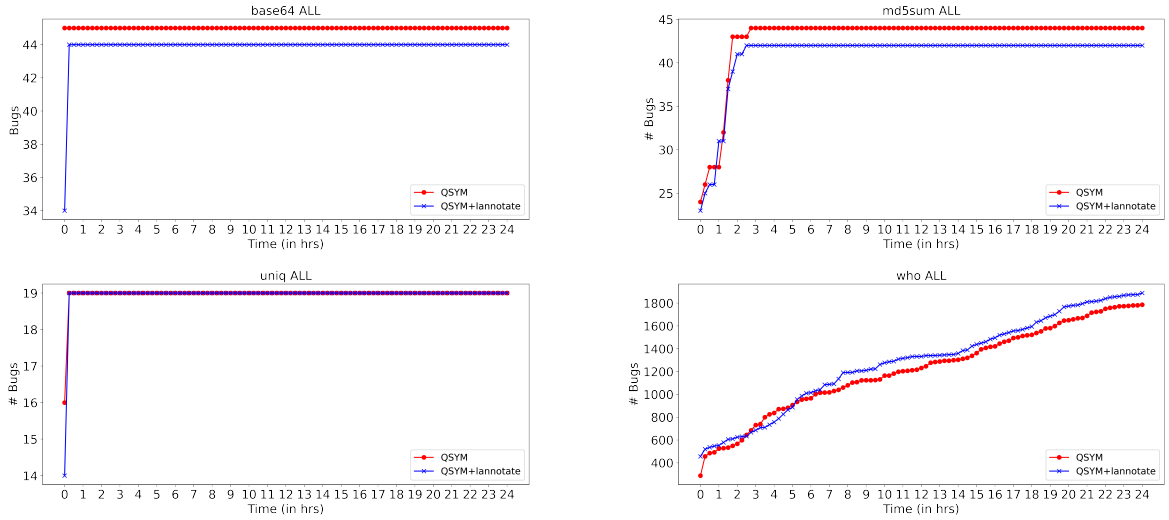


Fig. 9: Average number of bugs found by QSYM and QSYM+*lannot*

fuzzer embedded with a multi-level coverage metric enabling seed clustering. The key idea is to use finer-grained metrics such as edge coverage for seed selection and coarser-grained metrics such as block coverage for clustering. Furthermore, the technique uses a reinforcement learning-based hierarchical scheduler for seed selection. Contrary to these two works, our method does not alter the fuzzer to handle additional metrics, but it adds branches encoding the new objectives directly to the tested program, enabling off-the-shelf reuse of any fuzzer based on branch coverage. In addition, these two works focus

TABLE III: The absolute number of labels triggered by fuzzers on the LAVA-M benchmark suite. The *min*, *avg*, and *max* is the minimum, average and maximum number of labels covered. The number in the bracket shows the difference in comparison to the baseline.

App		AFL++	AFL++ +lannotate	QSYM	QSYM+lannotate
uniq	min	60	60	63	63
	avg	60	60	62	63 (+1)
	max	60	60	63	63
base64	min	25	25	27	26 (-1)
	avg	25	25	27	27
	max	25	25	27	28 (+1)
md5sum	min	95	89(-6)	94	95 (+1)
	avg	99	97(-2)	97	96 (-1)
	max	101	101	102	101 (-1)
who	min	82	82	78	82 (+4)
	avg	84	83 (-1)	80	85 (+5)
	max	88	88	84	88 (+4)

either on shallow standard metrics, like block and branch coverage, or on their own ad-hoc metrics, while our work proposes to leverage the many fine-grained metrics widely studied in the software testing literature. However, our method will also increase the number of inputs taken as seeds and may thus hamper the performance of the fuzzer for this reason as well. An analysis of the precise impact of our technique on fuzzer throughput is left for future work.

Ankou [32] proposes to modify the fuzzer to save as seed any input that covers any uncovered combinations of branches, instead of any uncovered branches alone. In practice, this means that the fuzzer uses some variant of the path coverage metric, instead of branch coverage. However, such a heavyweight metric delivers far too much data, resulting in seed explosion. Ankou reduces the amount of data with Principal Component Analysis (PCA) and performs adaptive seed pool updates to prevent seed explosion. In contrast, our technique enables supporting a wide set of additional metrics (and not just one) without modifying existing fuzzers. In addition, the metrics that we currently support are finer-grained than branch coverage but coarser-grained than path coverage, possibly providing a better compromise between fuzzing precision and seed explosion.

Fioraldi et al. [33] consider to save as seeds any input that violates likely block-level invariants of the program, collected through a prior dynamic analysis. The technique considers thus a very different source of information than ours to guide the fuzzer and, again, it requires the modification of the fuzzer to capture the violations of invariants.

Finally, a blog post [34] proposes to help fuzzers penetrate blocks guarded by a magic bytes comparison, through splitting the guard into nested smaller comparisons. While this mechanism also relies on additional branches in the code to guide the fuzzer, its essence is splitting a hard-to-penetrate branch into an equivalent sequence of easier-to-penetrate branches. Our approach is different, in the sense that we consider adding extra branches everywhere needed in the program to improve the guidance. We use this general mechanism to guide the fuzzer with state-of-the-art finer-grained coverage metrics and not to help it circumvent magic bytes comparisons.

B. More efficient coverage-guided fuzzing

Since the seminal publication of AFL [4], researchers have actively contributed to making fuzzing even more effective and efficient. In addition to the works discussed in the previous section, the most related to our proposal are the techniques that improve the initial selection of seeds or use a form a program transformation. Rebert et al. [35] mathematically formulate seed selection to maximise bugs discoveries. Herrera et al. [36] systematically investigate and evaluate the effects of initial seed selection strategies on bug finding. T-Fuzz [39] proposes a lightweight dynamic tracing-based technique to detect complex checks and bypass those checks by program transformation. The transformation is done by simply flipping the direction in the condition of the jump instruction. This approach leads to over-approximation and false positives. To address this, T-Fuzz leverages a symbolic execution-based approach to filter out false positives and reproduce true bugs in the original program.

IX. Conclusion

In this work, we have taken advantage of the large body of research over fine-grained code coverage criteria (such as multiple conditions and mutation coverage) and used these criteria as better proxies to select interesting inputs for mutation in coverage-based fuzzers. A noticeable aspect of our approach is that we make coverage-based fuzzers support most fine-grained coverage criteria out of the box (i.e., without changing their internals). We have achieved this by making the test objectives defined by these metrics (such as conditions to activate and mutants to kill) explicit as new branches in the target program. Fuzzing such a modified target is then equivalent to fuzzing the original target, but the fuzzer will also retain inputs covering the additional metrics objectives for mutation. We have proposed a preliminary evaluation of this novel idea using two state-of-art fuzzers, i.e., AFL++(3.14c) and QSYM with AFL(2.52b) on the four standard LAVA-M benchmarks. With one of the four Lava-M programs (*who*), we observe significantly better fuzzing results after transforming the program, with, on average, 100 more bugs being discovered in total and bugs being uncovered faster during the fuzzing process. With the three other Lava-M programs, the results indicate marginally worse results after transforming the program. While the positive results obtained with the *who* program encourage us to explore even more the potential of our approach, we have also discussed the main directions to achieve a strong and complete evaluation of it in the future, and we call for early feedback from the fuzzing community. These directions involve fast pruning of infeasible test objectives before fuzzing, extensive evaluation with more diverse benchmarks, discriminating the impact of specific coverage criteria, using more standard fuzzer evaluation metrics, as well as measuring impact over fuzzer throughput.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. This project has received funding from the European Union Horizon 2020 research and innovation program under grant agreement No 101021727 and from the France FUI CAESAR project.

REFERENCES

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, dec 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[2] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 627–637. [Online]. Available: https://doi.org/10.1145/3106237.3106295

[3] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *NDSS*, Feb. 2017.

[4] "american fuzzy lop - a security-oriented fuzzer," https://github.com/google/AFL, accessed: 2021-12-12.

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

[6] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USA: USENIX Association, 2018, p. 745–761.

[7] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," 01 2016.

[8] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," 02 2017.

[9] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. USA: Cambridge University Press, 2008.

[10] S. Bardin, N. Kosmatov, M. Marcozzi, and M. Delahaye, "Specify and measure, cover and reveal: A unified framework for automated test generation," *Science of Computer Programming*, vol. 207, p. 102641, 03 2021.

[11] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," 05 2016.

[12] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[13] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, 12 2018.

[14] "libFuzzer – a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, accessed: 2021-12-27.

[15] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2019.

[16] P. Ammann, J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, 2003, pp. 99–107.

[17] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[18] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[19] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[20] S. Bardin, O. Chebaro, M. Delahaye, and N. Kosmatov, "An all-in-one toolkit for automated white-box testing," vol. 8570, 07 2014.

[21] S. Bardin, N. Kosmatov, and F. Cheynier, "Efficient leveraging of symbolic execution to advanced coverage criteria," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 173–182.

[22] M. Marcozzi, M. Delahaye, S. Bardin, N. Kosmatov, and V. Prevosto, "Generic and effective specification of structural test objectives," in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, 2017, pp. 436–441.

[23] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[24] M. Marcozzi, S. Bardin, N. Kosmatov, M. Papadakis, V. Prevosto, and L. Correnson, "Time to clean your test objectives," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 456–467. [Online]. Available: https://doi.org/10.1145/3180155.3180191

[25] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. Le Traon, and J.-Y. Marion, "Sound and quasi-complete detection of infeasible test requirements," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.

[26] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 936–946.

[27] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, nov 2020. [Online]. Available: https://doi.org/10.1145/3428334

[28] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[29] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1597–1612.

[30] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 1–15. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/wang

[31] J. Wang, C. Song, and H. Yin, "Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing," in *NDSS*, 2021.

[32] V. J. M. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1024–1036. [Online]. Available: https://doi.org/10.1145/3377811.3380421

[33] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2829–2846. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[34] "laf-intel," https://lafintel.wordpress.com/, accessed: 2016-08-16.

[35] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 861–875.

[36] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 230–243. [Online]. Available: https://doi.org/10.1145/3460319.3464795