Fine-Grained Coverage-Based Fuzzing

WEI-CHENG WU, Université Paris-Saclay, CEA, List, France and University of Southern California, USA
BERNARD NONGPOH, Université Paris-Saclay, CEA, List, France
MARWAN NOUR, Université Paris-Saclay, CEA, List, France
MICHAËL MARCOZZI, Université Paris-Saclay, CEA, List, France
SÉBASTIEN BARDIN, Université Paris-Saclay, CEA, List, France
CHRISTOPHE HAUSER, University of Southern California, USA

Fuzzing is a popular software testing method that discovers bugs by massively feeding target applications with 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 interesting inputs to mutate. This work aims at taking advantage of the large body of research over defining finer-grained code coverage metrics (such as control-flow, data-flow or mutation coverage) and at evaluating how fuzzing performance is impacted when using these metrics to select interesting inputs for mutation. We propose to make branch 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 conditions to activate or 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. In addition, all the fuzzer mechanisms to penetrate hard-to-cover branches will help covering the additional metrics objectives. We use this approach to evaluate the impact of supporting two fine-grained coverage metrics (multiple condition coverage and weak mutation) over the performance of two state-of-the-art fuzzers (AFL++ and QSYM) with the standard LAVA-M and MAGMA benchmarks. This evaluation suggests that our mechanism for runtime fuzzer guidance, where the fuzzed code is instrumented with additional branches, is effective and could be leveraged to encode guidance from human users or static analysers. Our results also show that the impact of fine-grained metrics over fuzzing performance is hard to predict before fuzzing, and most of the time either neutral or negative. As a consequence, we do not recommend using them to guide fuzzers, except maybe in some possibly favorable circumstances yet to investigate, like for limited parts of the code or to complement classical fuzzing campaigns.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: fuzzing, code coverage criteria, mutation testing

ACM Reference Format:
Wei-Cheng Wu, Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin, and Christophe Hauser. 2023. Fine-Grained Coverage-Based Fuzzing. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2023), 40 pages. https://doi.org/10.1145/3587158

Authors' addresses: Wei-Cheng Wu, wwu@isi.edu, Université Paris-Saclay, CEA, List, France and University of Southern California, USA; Bernard Nongpoh, bernard.nongpoh@gmail.com, Université Paris-Saclay, CEA, List, France; Marwan Nour, marwan.s.nour@gmail.com, Université Paris-Saclay, CEA, List, France; Michaël Marcozzi, michael.marcozzi@cea.fr, Université Paris-Saclay, CEA, List, France; Sébastien Bardin, sebastien.bardin@cea.fr, Université Paris-Saclay, CEA, List, France; Christophe Hauser, hauser@isi.edu, University of Southern California, USA.

1049-331X/2023/1-ART1 $15.00
https://doi.org/10.1145/3587158

1 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 usual motive is to detect bugs as early as possible, before they cause failures or get exploited as vulnerabilities in production [3]. So called grey-box fuzzing has gained much attention in recent years. Grey-box fuzzers typically use a mutation- and coverage-based approach to generate new inputs from the ones generated before (Section 3). As inputs are being generated, those that cover yet uncovered branches of the PUT are saved as seeds and randomly mutated (i.e. slightly modified) to generate novel inputs, possibly exploring even more the newly discovered branches of the PUT. American Fuzzy Lop (AFL) and its community-maintained successor AFL++ [4, 5] are some of the most used and forked tools relying on such an approach. They are reported to have discovered many new CVEs within a wide range of applications in the recent years. Researchers and practitioners have proposed various methods to improve the raw input generation process of grey-box fuzzers. Notably, a strong limitation of raw grey-box fuzzing is that it struggles to cover the branches of the code that are guarded by difficult conditions, which are only activated by few specific "magic" input values. Two main solutions have been proposed to overcome this limitation. The first solution involves a form of *taint tracking*. For example, in *input-to-state correspondence* [6], constant values that appear in branching comparisons are collected and approximately tracked back to the PUT's inputs, in order to guide seeds mutations. Input-to-state correspondence has been recently implemented into AFL++. The second solution is *hydrid fuzzing*, which combines grey-box fuzzing with a program analysis called symbolic execution [7]. Symbolic execution has constraint solving capabilities able to systematically discover the magic input values to enter the difficult branches. These values are then used as mutation seeds by the grey-box fuzzer to explore the resolved branches. QSYM [8] is one of the most popular hydrid fuzzing tools proposed so far.

Problem. Branch coverage is a shallow metric to evaluate how well the possible behaviours of a program are exercised. As a consequence, software testing researchers have defined standard coverage metrics that are finer-grained than simply counting branches [9] (Section 3), such as multiple condition coverage or mutation coverage. By making fuzzers more sensitive in retaining inputs that trigger new behaviours of the program, one may hope that using such fine-grained metrics for seed selection would make PUT exploration more effective. Yet, these fine-grained metrics are not used in state-of-the-art fuzzers, which rely solely either on some forms of branch coverage (also known as edge coverage or decision coverage), or on ad hoc mechanisms [10–12] for guidance. The fuzzing community has just started to investigate ways to support some specific finer-grained coverage metrics within specific fuzzers (Section 7) and the general ability of such metrics to actually improve fuzzing in practice remains unknown.

Goals and challenge. In this work, we intend to challenge the position of branch coverage as the de facto guidance metric for fuzzing. To do so, we aim at (1) providing an effective and generic means to support fine-grained coverage metrics in state-of-the-art fuzzers and (2) evaluating the impact of using such metrics (in addition to branch coverage) over the ability of fuzzers to exercise the PUT and find bugs in it. A significant challenge to overcome is harnessing the wild variety of fuzzer implementations and of fine-grained metrics to support. In particular, we do not want to require digging into the internals of every fuzzer implementation and finding a way to extend them with ad hoc support for every additional criterion.

Proposal. We propose to make state-of-the-art 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 2 and 4). 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 [13]. This makes it possible, given a PUT and a fine-grained coverage metric, to carefully instrument the code of the PUT with new branches corresponding to the objectives from the metric, without modifying the PUT's semantics. Covering one of these branches is then equivalent to covering the corresponding objective from the metric. Fuzzing such a transformed PUT with any coverage-based fuzzer (relying 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. In addition, all the mechanisms implemented by the fuzzer to penetrate difficult branches (like taint tracking and hybrid fuzzing) will serve for free to help cover difficult fine-grained objectives. Finally, comparing the performance of the fuzzer on the original and transformed PUT makes it possible to measure the impact of using the metric over the ability of the fuzzer to exercise the PUT and find bugs in it.

Experiments. We chose to realise our proposal using the multiple condition coverage and weak mutation coverage metrics. These representative fine-grained metrics are known to be stronger than branch coverage [14–16]. 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 common programming mistakes in the program corrupt the program state. We develop a tool that follows our proposal to automatically instrument C code with the objectives from these two fine-grained coverage metrics (Section 5). We use this tool to instrument twelve various programs from LAVA-M [17] and MAGMA [18], two standard fuzzing benchmarks, totalling more than seven hundreds of thousands of lines of code. We repeatedly run the state-of-the-art AFL++ grey-box fuzzer (with input-to-state correspondence) and QSYM hybrid fuzzer over the twelve original programs and their transformed versions (Section 6), for a total of two and a half years of CPU time. We measure and compare the fuzzing performance between the averaged runs, in terms of fuzzing throughput, covered branches and detected bugs. We also try and capture how this performance is affected, when carefully selecting which of the fine-grained objectives are instrumented into the programs, e.g. by using static analysis to prune out beforehand any infeasible objectives.

Findings. The profitability of a fine-grained coverage metric over a PUT is roughly the difference between the gain (or penalty) that it provides to fuzzer guidance and the fuzzer slowdown that it causes by adding more objectives to monitor. Our experiments show that the measured profitability of the evaluated metrics is hard to predict in advance for a given PUT and most of the time either neutral or negative in practice. Notably, we have observed a better profitability when the PUT was loaded with a high density of bugs, while the metric had produced a not too high density of fine-grained objectives. Indeed, fine-grained coverage objectives enable a denser sampling of the subtle local differences of behaviour in the code, probably at the expense of a broader coverage of the complete codebase. As a consequence, they seem good at helping fuzzers clean bug nests (high bug density), but bad at helping them find a needle in a haystack (low bug density). Moreover, a too high density of fine-grained objectives added to the PUT appears to reduce the fuzzer's throughput enough to worsen the fuzzing results (by making the PUT slower to run and by increasing the amount of coverage data to process). Yet, concretely, it seems difficult to predict the profitability of a fine-grained metric over a given PUT before actually fuzzing its instrumented version. As a consequence, and considering that positive profitability looks infrequent in practice, we do not recommend fine-grained coverage metrics as a general means to guide fuzzers. We feel that this recommendation is strengthened by the fact that the faced issues appear similar to those reported in simultaneous works [19–21] (Section 7), where a state-of-the-art fuzzer is modified to support a single fine-grained coverage metric. Still, it remains an open question whether such metrics could be useful in some favorable use cases, like instrumenting only sensitive or fragile parts of the codebase, or running additional fuzzing campaigns, to complement traditional ones based on branch coverage. Finally, the performed experiments also suggest that the proposed mechanism to support additional fuzzing objectives, without modifying the fuzzer but by adding new dedicated branches in the PUT, is effective. In addition to encode objectives from fine-grained coverage metrics, this mechanism could be used in other contexts where additional guidance is to be provided to fuzzers at runtime, like with human directives or bug preconditions computed by static analysers.

Contributions. To sum up, the two main contributions of this paper are:

(1) An effective mechanism to guide at runtime the behaviour of existing fuzzers based on branch coverage. This mechanism involves instrumenting the fuzzed code with additional branches, making the fuzzer keep as seeds the inputs that unlock the conditions crafted to guard the entrance of these branches. Implementing this approach notably requires a careful design (which we detail in the paper) of the guarding conditions and branch bodies, to avoid them either breaking the semantics of the instrumented program, or making the fuzzer report spurious crashes, or being tampered by the compilation and harnessing of the program. The proposed instrumentation mechanism can be used to encode objectives from fine-grained coverage metrics where needed in the fuzzed code, but it has also the potential to serve as a *lingua franca* for any additional guidance to be provided to fuzzers at runtime, like human directives or bug preconditions from static analysers. For any program state deemed interesting by the guidance but hard to reach by the fuzzer, our approach enables taking advantage for free from all the means already embedded in fuzzers to enter difficult branches;

(2) An experimental evaluation of the impact of supporting two representative fine-grained coverage metrics (multiple condition coverage and weak mutation) over the performance of a state-of-the-art grey-box and hybrid fuzzer (AFL++ and QSYM), while fuzzing more than seven hundreds of thousands of lines of code from the standard LAVA-M and MAGMA benchmarks, for more than two years and a half of CPU time. This evaluation reveals that the impact of such metrics is hard to predict before fuzzing and most of the time either neutral or negative. These results suggest thus that fine-grained metrics should not be used as a general means to guide fuzzers. Yet, they also asks the question of whether such metrics could be useful in some specific favorable circumstances, like for limited parts of the codebase or as a complement to classical fuzzing campaigns.

Overall, as the interest in fine-grained coverage metrics is rising in the fuzzing community, we provide in this work a significant step towards better understanding how these metrics could or could not be useful in the context of fuzzers.

Artifact. Our open-source tool and experimental infrastructure are available as an artifact[1].

2 MOTIVATING EXAMPLE

We illustrate now with a simple example how our approach can make state-of-the-art fuzzers support fine-grained coverage metrics 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 above its allowed temperature limit and taking corrective actions if so. We suppose that there is a bug in the conditional checking that the temperature limit is exceeded

[1]https://git.frama-c.com/bnongpoh/cannotate (tool) and https://doi.org/10.5281/zenodo.7133734 (infrastructure)

```
void check(int current_temp,char *data[] ){
if(current_temp>=50) // Bug: should be current_temp>50
    {
    // Deal with appliance running above the allowed temperature limit
    ...
        // The bug triggers a detectable crash only when current_temp==50
        // and when rare specific values are present in data
    }
}
```

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

```
void check(int current_temp,char *data[] ){
...
if (current_temp>=50 != current_temp==50) { } // Exact condition to kill mutant #1
if (current_temp>=50 != current_temp>50) { // Exact condition to kill mutant #2
/* Keeping and mutating inputs entering here helps the fuzzer triggering the crash */ }
if(current_temp>=50) // Bug: should be current_temp>50
    {
    // Deal with appliance running above the allowed temperature limit
    ...
        // The bug triggers a detectable crash only when current_temp==50
        // and when rare specific values are present in data
    }
}
```

Listing 2. Same program as Listing 1, but with instrumentation for fine-grained fuzzing with the Weak Mutation criterion (ROR operator).

(it should be `current_temp>50` instead of `current_temp>=50`), but that this bug only triggers a program crash (enabling a fuzzer to detect it) when some rare values are provided in the `data` argument (requiring the fuzzer to generate many inputs making `current_temp` equal to 50 to actually trigger it).

The Weak Mutation Coverage criterion is a fine-grained code coverage metric requiring the inputs to trigger run-time behaviours that differentiate the PUT from a set of mutant versions of it. Such mutants are slight syntactic variants of the PUT, built by planting some common patterns of programming mistakes in it. When an input makes the program execution diverge between the PUT and one of its mutant, this input is said to cover, or "kill", the mutant. Our approach can make the exact condition to kill each mutant explicit in the code of the PUT. We illustrate this in Listing 2 by adding two conditional statements, corresponding respectively to killing two of the mutants produced using a state-of-the-art mutant creation algorithm (Relational Operator Replacement, a.k.a. ROR, which seeds faults by switching comparison operators). The transformed program can then be fuzzed using an off-the-shelf coverage-based fuzzer relying on branch coverage. When this coverage-based fuzzer will produce an input entering the then branch of one of these new conditionals (and thus killing the corresponding mutant), it will save it for mutation[2].

It should be remarked that the second added conditional explicitly forces the fuzzer to maintain and mutate an input where `current_temp` is equal to 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.

[2]Note that we use the word "mutation" to refer both to the Mutation code coverage criteria and to the seed mutations performed by fuzzers. We have kept this ambiguity in the paper to be consistent with the common practice in both the mutation testing and fuzzing communities.

3 BACKGROUND

3.1 Coverage-based fuzzing

Grey-box fuzzers typically use a coverage-based feedback mechanism to improve the efficiency and effectiveness of the input generation process. Figure 1 shows the general working principle of such a grey-box fuzzer. It starts with an initial set of user-provided input seeds; if unavailable, the fuzzer will construct one [22][23] 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 using

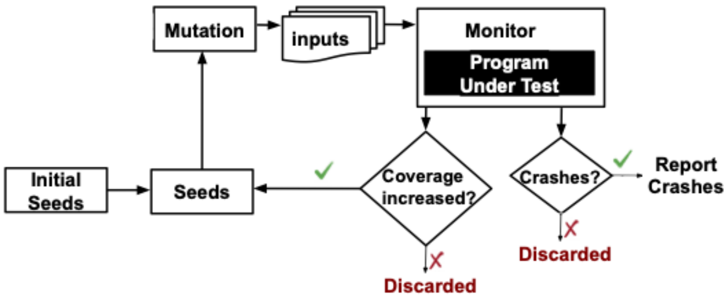


Fig. 1. General coverage-based grey-box fuzzing process

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. A highly popular implementation of a grey-box fuzzer is AFL [4] / AFL++ [5].

Hyrbid fuzzers augment the grey-box fuzzing process with an additional source of seeds. Such fuzzers indeed use a symbolic execution engine [7] to systematically construct seeds able to penetrate the branches that the fuzzer has little chance to satisfy, using its random trial-and-error exploration of the input space (e.g. penetrating `if (input == 0xdeadbeef)` would require in average 2^{31} trials). Popular implementations of a hybrid fuzzer are Driller [24] and QSYM [8].

3.2 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 [25] [26] [27], 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 PUT;
- *Decision Coverage (DC), similar to branch coverage or edge coverage:* requires a test suite to activate both the true and false path of each decision point in the program under test. This is similar to covering all the edges in the control-flow graph of the program. Estimating in some way the number of covered control-flow edges is 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* refer to the building blocks of logical expressions, which are logical expressions on their own, but which can also be combined with each others using logical operators (like conjunction, disjunction or negation), in order to build compound logical 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.

At this moment, it should be noted that *most research works over code coverage criteria implicitly define coverage w.r.t. the source code of the PUT, written in a high-level programming language. On the contrary, many fuzzers, like AFL and AFL++, compute (decision) coverage at assembly level, which, considering common compilation approaches, is close, but not perfectly equivalent, to doing it at source level.* Notably, in the C/C++ programs usually considered in fuzzing research, logical expressions at decision points often rely on lazy logical operators (&& or ||). Such expressions are usually compiled into assembly code doing a short-circuiting evaluation of their atomic conditions, which adds (nested) decisions in the assembly for each of these atomic conditions. When the assembly includes such additional nested decisions, one can check that AFL's DC at assembly level is a stronger criterion than both DC and CC at source level (i.e. a test suite satisfying AFL's DC at assembly level will satisfy both DC and CC at source level, while a test suite satisfying either DC or CC at source level may require additional tests to satisfy AFL's DC at assembly level). Yet, MCC at source level remains a stronger criterion than AFL's DC at assembly level (because a test suite satisfying AFL's DC at assembly level may not cover all the possible *combinations* of atomic truth values). As a consequence, using (source-level) CC to pilot a fuzzer like AFL should provide only a limited additional guidance compared to vanilla AFL, while using MCC should be a better guide[3].

Mutation criteria. Mutation criteria are derived from the research efforts in mutation testing [28]. Test objectives consist here of mutants, i.e. slight variants of the program under test, seeded with common faults. 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 difference is observable in the internal states of the PUT and mutant around the mutation points, we say we say that the input *weakly* kills the mutant. If this this difference is also observable from outside of the code, as the PUT and the mutant generate different outputs, we say that the input *strongly* kills the mutant. From these definitions, one can see that killing a mutant weakly is necessary to kill it strongly, but that an input killing a mutant weakly may not kill it strongly, if the divergence between the internal states does not propagate to the outputs. From this, we can define the strong and weak mutation coverage criteria:

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

Empirical results indicate that "weak mutation can be applied in a manner that is almost as effective as [strong] mutation testing, and with significant computational savings" [29].

In order to create a significant set of mutants, seeded with classical faults, for a given PUT, one should use standard mutant creation operators, like Absolute Value Insertion *(ABS)*, Arithmetic

[3]This is what motivates the fact that MCC will be one of the fine-grained metrics used to guide fuzzers in this work.

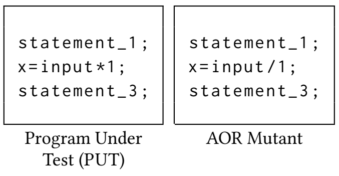
Operator Replacement *(AOR)*, Conditional Operator Replacement *(COR)* and Relational Operator Replacement *(ROR)* for programs written in simple imperative style.

Data-flow criteria. Data-flow criteria require a test suite to cover a chosen subset of the variable definition-use pairs in the PUT. For a given variable v in the PUT, a definition-use pair for v couples a statement setting or modifying the value of v (definition) with a subsequent statement using of the value of v (use). The definition and the use are linked by a definition-clear path if there exists at least one control-flow path from the definition to the use such that v is not modified in-between. An input covers a definition-use pair for v if running the PUT with this input traverses at least once a definition-clear path linking the pair together. From this, we can define the common dataflow coverage criteria:

- *All definitions coverage (all-def):* requires a test suite to cover at least one definition-use pair for every variable definition in the PUT;
- *All uses coverage (all-use):* requires a test suite to cover at least one definition-use pair for every variable use in the PUT;
- *All definition-use pairs coverage (all-def-use):* requires a test suite to cover all the definition-use pairs for every variable in the PUT.

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.

Infeasible coverage objectives. As one might have already noticed, code coverage criteria are defined in a purely syntactic way and thus totally blind to the semantics of the program under test. As a consequence, some of the test objectives that they define in the PUT (i.e. truth values to activate, mutants to kill, definition-use pair to cover) may turn out to be *infeasible* in practice, i.e. no input can actually satisfy them. A well-known example of such infeasible test objectives is equivalent (a.k.a. immortal) mutants, i.e. mutants that are semantically equivalent to the PUT, so that they cannot be killed by any possible input. Figure 2 illustrates how a mutant built using the standard (and purely syntactic) AOR operator can turn out to be infeasible in practice.



```
statement_1;
x=input*1;
statement_3;
```

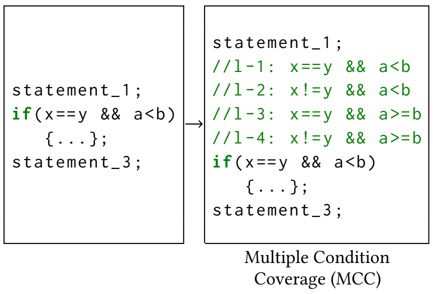
```
statement_1;
x=input/1;
statement_3;
```

Fig. 2. Equivalent (i.e. immortal/infeasible) mutant created with the AOR operator.

The main issue with infeasible test objectives is that test suite builders often do not know whether they fail to cover some objectives because their test suite is weak, or because these objectives are infeasible. As a consequence, they may possibly waste a significant amount of their test budget trying to find additional test inputs to cover objectives that cannot. Several works have tried detect and prune out as many of such infeasible objectives as possible automatically (the problem is generally not decidable), mainly by using a static analysis, either from a dedicated analyser [30] or by diverting those of a standard compiler [31], pruning out up to more than 7% of the objectives as infeasible.

3.3 Making test objectives explicit with labels

Bardin et al. [13] [32] [33] have proposed a generic mechanism for specifying the test objectives from many coverage criteria explicitly within the (source) code of the PUT. 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 3 and 4 illustrate how the test objectives from the 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 (e.g. dataflow or strong mutation) explicit. Marcozzi et al. [34] discuss this issue and extend the expressiveness of labels into hyperlabels, to handle a wider set of coverage criteria. Alternatively, support for some non-encodable criteria can also be provided with labels, by adding helper code and variables to the PUT, to be used in the label conditions. This last approach is illustrated in Figure 5, where the test objectives of the all definition-use pairs data-flow criterion are made explicit with raw labels, thanks to helper code to monitor definition-clear paths.

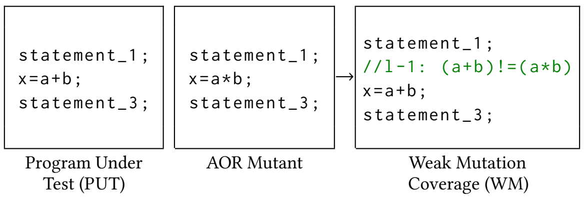


```
statement_1;
if(x==y && a<b)
   {...};
statement_3;
```

→

```
statement_1;
//l-1: x==y && a<b
//l-2: x!=y && a<b
//l-3: x==y && a>=b
//l-4: x!=y && a>=b
if(x==y && a<b)
   {...};
statement_3;
```

Fig. 3. Encoding MCC test objectives with labels [13].



```
statement_1;
x=a+b;
statement_3;
```

```
statement_1;
x=a*b;
statement_3;
```

→

```
statement_1;
//l-1: (a+b)!=(a*b)
x=a+b;
statement_3;
```

Fig. 4. Encoding WM test objectives with labels (considering a single mutant created with the AOR operator).

4 FINE-GRAINED COVERAGE-BASED FUZZING

4.1 General principle

Given the (source) code of a program P and a label-encodable code coverage criterion[4] 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.

[4]While our approach, as described in the current section, is independent of the chosen code coverage criterion (as long as it is encodable with labels), our current implementation of this approach, as described and evaluated in the subsequent sections, provides support only for the MCC and WM criteria.

```
int flag = 0;
if (do_action()) {
    flag = 1;
}
report_error_if_non_zero(flag);
```

Program Under Test (PUT)

→

```
int flag = 0;
/* Helper line */ int def_flag_1 = 1;
if (do_action()) {
    //l-1: def_flag_1
    flag = 1;
    /* Helper line */ def_flag_1 = 0;
}
//l-2: def_flag_1
report_error_if_non_zero(flag);
/* Helper line */ def_flag_1 = 0;
```

All Definition-Use Pairs Coverage (all-def-use)

Fig. 5. Encoding data-flow test objectives with labels (with helper code to monitor definition-clear paths).

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 I 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 6. 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 I, the fuzzer will basically save as a seed an input covering l and its corresponding objective from C.

```
statement_1;
//l-1: x==y && a<b
//l-2: x!=y && a<b
//l-3: x==y && a>=b
//l-4: x!=y && a>=b
if(x==y && a<b)
   {...};
statement_3;
```

P with labels for MCC

→

```
statement_1;
if (x==y && a<b) {}
if (x!=y && a<b) {}
if (x==y && a>=b) {}
if (x!=y && a>=b) {}
if(x==y && a<b)
   {...};
statement_3;
```

P_{lannot} for MCC

Fig. 6. Transforming a program for fine-grained coverage-based fuzzing with MCC criterion [13].

4.2 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 7, where the transformed program would typically print "abaabaab" instead of just "ab", like in the original program.

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 affecting the label predicates, without modifying the semantics of P. This normalization process is illustrated in Figure 8 for side-effects appearing in the atomic conditions of decision points in the program (such conditions are affecting 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

```
statement_1;
//l-1: print("a") && print("b")
//l-2: !print("a") && print("b")
//l-3: print("a") && !print("b")
//l-4: !print("a") && !print("b")
if(print("a") && print("b"))
   {...};
statement_3;
```

P with labels for MCC

→

```
statement_1;
if (print("a") && print("b")) {}
if (!print("a") && print("b")) {}
if (print("a") && !print("b")) {}
if (!print("a") && !print("b")) {}
if(print("a") && print("b"))
   {...};
statement_3;
```

P_{lannot} for MCC

Fig. 7. Naive transformation of a C program containing expressions with side-effects.

because of a lazy boolean operator. Case (b) and (c) detail the conditional extraction performed in this situation, respectively for a lazy && and || operator[5].

```
if (print("a")) {/* then block */}
else {/* else block */}
```

→

```
int temp = print("a");
if (temp) {/* then block */}
else {/* else block */}
```

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

```
if (print("a") && print("b")) {/* then block */}
else {/* else block */}
```

→

```
int temp_a = print("a");
if (temp_a) {
        int temp_b = print("b");
        if (temp_b) {/* then block */}
        else goto label_1;
} else { label_1: /* else block */}
```

(b) Side-effect in the operands of a lazy boolean operator (AND case)

```
if (print("a") || print("b")) {/* then block */}
else {/* else block */}
```

→

```
int temp_a = print("a");
if (temp_a) {
        goto label_1;
} else {
        int temp_b = print("b");
        if (temp_b) {label_1: /* then block */}
        else {/* else block */}
}
```

(c) Side-effect in the operands of a lazy boolean operator (OR case)

Fig. 8. Extracting side-effects from atomic conditions at program decision points.

4.3 Preventing label instrumentation from introducing spurious runtime errors

Transforming $P_{normalised}$ into P_{lannot} adds thus empty conditional statements for all the test objectives required by the criterion C, to a semantically equivalent but normalised version of P, where expressions with side-effects have been pre-processed. Yet, despite this pre-processing, instrumenting $P_{normalised}$ with the empty conditionals can still alter the semantics of the code, in case executing these conditionals would trigger crashes on some inputs, while $P_{normalised}$ would run normally on these inputs. This can actually happen quite easily if one instruments the code naively, and it will cause the fuzzer to report many spurious crashes in the instrumented program. For example, it is a common programming practice that "dangerous" expressions appearing in a program, like pointer accesses or number divisions, are guarded by defensive code aimed at avoiding their evaluation with concrete values leading to a crash (like a null pointer or a zero

[5]In this example, we consider the semantics of C where the operands of a lazy boolean operator are evaluated from left to right. More precisely, if the value of the first operand is sufficient to determine the result of the operation, the second operand is not evaluated.

divisor). Figure 9 illustrates how naively instrumenting code from $P_{normalised}$ containing such guarded expressions can lead to the dangerous expression being evaluated unguarded within the instrumentation code, possibly triggering spurious crashes at runtime.

```
statement_1;
//l-1: pointer==NULL && !*pointer
//l-2: pointer==NULL && *pointer
//l-3: pointer!=NULL && !*pointer
//l-4: pointer!=NULL && *pointer
if(pointer!=NULL && *pointer)
   {...};
statement_3;
```

$P_{normalised}$ with labels for MCC

$\rightarrow$

```
statement_1;
if (pointer==NULL && !*pointer) {/* SegFault */}
if (pointer==NULL && *pointer) {/* SegFault */}
if (pointer!=NULL && !*pointer) {}
if (pointer!=NULL && *pointer) {}
if(pointer!=NULL && *pointer)
   {...};
statement_3;
```

P_{lannot} for MCC

```
statement_1;
//l-1: (666/divisor) != (666+divisor)
//l-2: (666/divisor) != (666-divisor)
//l-3: (666/divisor) != (666*divisor)
if(divisor>0 && (number > 666/divisor))
   {...};
statement_3;
```

$P_{normalised}$ with labels for WM (AOR operator)

$\rightarrow$

```
statement_1;
if ((666/divisor) != (666+divisor)) {/* ZeroDiv */}
if ((666/divisor) != (666-divisor)) {/* ZeroDiv */}
if ((666/divisor) != (666*divisor)) {/* ZeroDiv */}
if(divisor>0 && (number > 666/divisor))
   {...};
statement_3;
```

P_{lannot} for WM AOR

Fig. 9. Naive transformation of programs leading to spurious runtime errors being introduced.

The easiest way to solve this issue is to catch and hide all the runtime errors produced by the conditional statements added by our instrumentation. This is possible in programming languages that provide built-in exception management and can be implemented by surrounding the conditionals by try-catch constructs. For languages that do not provide built-in exception management, like C, the predicates of the conditionals must be analysed statically, to derive a guarding condition that would prevent any crash during their evaluation. The conditionals are then guarded by this additional condition. It should be noted that writing a complete algorithm to compute this condition might be difficult in practice, if the semantics of the considered programming language is complex and not-fully documented, so that crash-triggering situations can be easily missed[6]. These two approaches to instrumenting the code of $P_{normalised}$ in a robust way against spurious runtime errors is illustrated at Figure 10.

4.4 Pruning out infeasible and trivial labels

Running the fuzzer on P_{lannot} in the unavoidable presence of empty conditionals encoding infeasible test objectives should have two negative effects:

(1) Increase the execution time of the program (and thus reduce the fuzzer's throughput) without providing any additional guidance to the fuzzer, as no input will ever enter these conditionals. (note that a similar issue will also happen with trivial test objectives, which yield conditionals with an always true condition).
(2) For hybrid fuzzers, the symbolic execution engine will be polluted by these conditionals, as they add paths with an unsatisfiable path constraint to the PUT, reducing the ability of symbolic execution to support grey-box fuzzing.

[6]In our implementation of our approach, which targets C programs, we have considered all the common crash-triggering situations for C that we were aware of. While we cannot exclude to have missed some cases, this has not resulted in unexplained crashes during the fuzzing campaigns that we have conducted.

```
// ...
//l-x: (666/divisor) !=
//     (666*divisor)
// ...
if(divisor>0 &&
   (number > 666/divisor))
   {...};
```

$P_{normalised}$
with labels

```
...
try {
if ((666/divisor) !=
    (666*divisor)) {}
} catch (ZeroDiv) {}
...
if(divisor>0 &&
   (number > 666/divisor))
   {...};
```

Robust instrumentation
(with exception handling)

```
...
if (divisor != 0 &&
    ((666/divisor)
    != (666*divisor))) {}
...
if(divisor>0 &&
   (number > 666/divisor))
   {...};
```

Robust instrumentation
(without exception handling)

Fig. 10. Robust label instrumentation against spurious runtime errors.

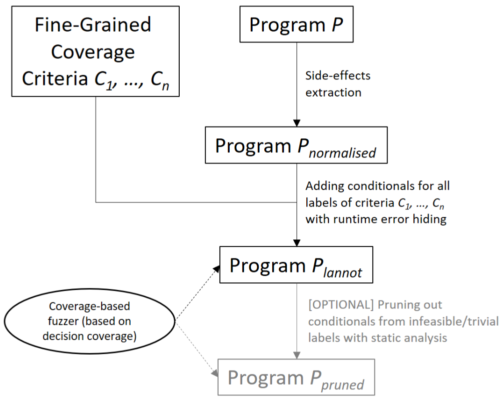

Fine-Grained Coverage Criteria C_1, ..., C_n
Program P
Side-effects extraction
Program $P_{normalised}$
Adding conditionals for all labels of criteria C_1, ..., C_n with runtime error hiding
Program P_{lannot}
Coverage-based fuzzer (based on decision coverage)
[OPTIONAL] Pruning out conditionals from infeasible/trivial labels with static analysis
Program P_{pruned}

Fig. 11. Fine-grained coverage-based fuzzing workflow

Yet, the static analysis that should be performed to try and prune out these objectives before fuzzing would eat a part of the fuzzing budget, while the actual impact of their removal on fuzzing performance is uncertain. As a consequence, we consider an optional step to our approach, where static analysis is used to remove infeasible (and trivial) labels from P_{lannot} before fuzzing (yielding P_{pruned} to be fuzzed), and aim at measuring the impact of this step on the overall bug detection performance, as a part of our experimental evaluation.

4.5 Complete workflow

Figure 11 summarises the complete workflow of our fine-grained coverage-based fuzzing approach. Note that while we have described the approach in the context of a single coverage criterion C, test objectives from several criteria $C_1, ..., C_n$ can also be mixed together in practice, in the hope of combining their guidance powers. After extracting side-effects from expressions able to impact the label predicates, we add conditionals corresponding to the labels making the criteria explicit. We use a robust instrumentation hiding the runtime errors that would occur while executing these conditionals. The transformed PUT is then fuzzed using a classical (decision 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 the additional objectives from the criteria. As an additional step, static analysis can be used to prune out the conditionals corresponding to infeasible and trivial objectives from the transformed PUT, before fuzzing it.

5 IMPLEMENTATION

We have implemented our program transformation approach for C programs as passes in the Clang open-source compiler infrastructure [35]. 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 the expressions that will subsequently be involved in the labels from the MCC or WM criteria, in case they contain side-effects (The Clang's AST provides the `HasSideEffects()` primitive to detect them). Our program annotation pass inserts robust conditional statements for the MCC and WM (ABS, AOR, COR and ROR operators) criteria, using the *Rewriter* class provided by the *Clang API*. The conditionals are made robust by analysing their predicate and prefixing unsafe operations (like pointer accesses and number divisions) with a guarding atomic condition (like checking that the pointer is not null and the divisor different from zero).

We have carefully inspected by hand the assembly code produced after (1) transforming a sample program with our passes, (2) instrumenting the transformed code with the fuzzer coverage measurement harness and (3) compiling the result with gcc, in order to make sure that our transformation was compatible with (and thus not tampered) by the harnessing or compilation processes. This inspection revealed that preventing such tampering requires deactivating aggressive compiler optimisations and adding the `asm volatile ("":: "memory");` statement in the body of the inserted conditional statements.

Our optional step, where infeasible/trivial conditionals are detected (and pruned out), is implemented by diverting the static analyses of a standard compiler (namely, gcc), in a similar way to Papadakis et al. [31]. More precisely, for each conditional, we compile the PUT with and without this conditional added into it. Before running the compiler, we insert a print statement in the body of the added conditional, so that this conditional cannot be optimised away, except if its branching condition is always false. The two binaries produced by the compiler are then compared. If they are identical, it means that the conditional was optimised away and is thus infeasible. Trivial conditionals can be detected using the same process, but with a negated branching condition.

6 EXPERIMENTAL EVALUATION

6.1 Objectives

We aim at collecting data to answer the following research questions:

RQ1 What is the impact of making the coverage objectives from fine-grained criteria explicit in the PUT, over the guidance of state-of-the-art grey-box and hybrid fuzzers? More precisely, how are the fuzzers' (1) throughput, (2) size of seeds pool, (3) achieved edge coverage and (4) number of detected bugs in the PUT, affected?

RQ2 Can the measured impact of fine-grained coverage objectives over fuzzer guidance be improved by carefully selecting which of these objectives are made explicit in the PUT, e.g. by pruning infeasible/trivial coverage objectives out?

6.2 Experimental setup

To collect data to answer our research questions, we use our code transformation tool to add conditional statements for MCC and WM objectives to the programs of two standard benchmarks used in fuzzing research, namely LAVA-M [17] and MAGMA [18]. LAVA-M involves small and single-file C programs, automatically seeded with 2,368 artificial bugs at a very high density (one bug every six lines of code in average). MAGMA involves large and multi-file C programs (sometimes to be compiled to a set of several independatant executables)[7], manually injected with 116 real bugs at a low density (one bug every 6k lines of code in average, i.e. three orders of magnitude less density than in LAVA-M). Both benchmarks provide built-in logging mechanisms, which report when the bugs are triggered during a fuzzing campaign (with a unique ID per bug to avoid double counting). Details about all the programs can be found in Table 1.

Table 1. Programs from standard fuzzing benchmarks used to evaluate our approach

Benchmark	**Principle**	**Program**	**Executables**	**Lines of code**	**Bugs**	**Bug density**
LAVA-M	Injecting artificial bugs in GNU coreutils	base64	base64	324	48	1 per 7 loc
		uniq	uniq	700	29	1 per 24 loc
		md5sum	md5sum	1,021	57	1 per 18 loc
		who	who	12,311	2,234	1 per 6 loc
		TOTAL		**14,356**	**2,368**	**1 per 6 loc**
MAGMA	Front-porting fixed bugs to latest version of open-source libraries	lua	lua	16,633	4	1 per 4,158 loc
		php	exif	22,733	16	1 per 1,421 loc
		libsndfile	sndfile	51,748	18	1 per 2,875 loc
		libpng	libpng_read	52,200	7	1 per 7,457 loc
		libtiff	tiff_read_rgba tiffcp	70,561	14	1 per 5,040, loc
		libxml2	read_memory xmlint	151,758	17	1 per 8,927 loc
		sqlite3	sqlite3	171,130	20	1 per 8,556 loc
		openssl	server client x509	188,442	20	1 per 9,422 loc
		TOTAL		**722,205**	**116**	**1 per 6,226 loc**

We consider six different settings for each executable. The first one is the original executable (baseline) and the five others are this executable with *some of the label-derived conditional statements added* (our approach). More precisely, the second one contains only the conditionals from the MCC criterion, the third only those from WM, the fourth both those from MCC and WM, the fifth a select subset of the MCC conditionals (how this subset is created will be explained in Section 6.5) and the sixth both those from MCC and WM, but without those deemed infeasible by static analysis.

We fuzz each executable in all settings using the most recent state-of-the-art *afl++3.14c* grey-box fuzzer[8] and *QSYM* hybrid fuzzer. Each of the resulting possible configurations, involving a particular fuzzer over a particular executable in a particular setting, is fuzzed during a 24-hours campaign. To mitigate the impact of randomness, we run five campaigns for each configuration. This leads to five 24h runs for 192 configurations, i.e. about two and a half years (23,040 hours) of CPU time, which were performed (one core per run) on two cloud servers, loaded each with two Intel Xeon Silver 4214 CPUs, with 377GB of RAM and 12 logical cores per CPU, running at up to 3.2GHz.

[7]MAGMA also includes one C++ program, poppler, which we do not consider here, as our instrumentation tool can only process pure C programs. We have also skipped some of the executables in the openssl (asn1, asn1parse and bignum) and php (unserialize, parser and json) programs, as the first ones were generating huge amounts of data (>100GB), making our servers crash, while no bug could be triggered with the second ones, both in our experiments and on the MAGMA website.
[8]All the experiments are performed using AFL++ with the "-m none -c -d" flags on (no memory limit, logging of comparison operands for clever mutations and skipping deterministic mutation operators). These flags enable AFL++ to deal with branches guarded by magic bytes comparisons through input-to-state correspondence [6] and thus find bugs in LAVA-M, which the original AFL fuzzer, extended by AFL++, can barely do.

6.3 Program instrumentation, build and sanity checks

Instrumenting the LAVA-M and MAGMA benchmark programs to make the MCC and WM coverage objectives explicit lead to 337,311 conditionals being inserted in the code. Considering that each conditional represents one additional line of code, the instrumentation increased the size of programs by 46% on average, varying between one conditional every line of code (in the libxml2 program) and one conditional every five lines of code (in the who program). Note that the instrumented program is aimed at being used during fuzzing campaigns only and not in production. A detailed breakdown of the conditionals insertion numbers can be found in Table 2. Manual inspection revealed that the different coding styles of the instrumented programs lead to different densities of coverage objectives of all types. For example, the huge number of MCC objectives for libxml2 is due to the common use of "one by one character comparisons in single condition guards" within the libxml2 codebase, e.g.

```
if ((cur == '<') && (next == '!') && (ctxt->input->cur[2] == 'D') && (ctxt->input->cur[3] == 'O') &&
    (ctxt->input->cur[4] == 'C') && (ctxt->input->cur[5] == 'T') && (ctxt->input->cur[6] == 'Y') &&
    (ctxt->input->cur[7] == 'P') && (ctxt->input->cur[8] == 'E')) { ... }
```

Table 2. Quantity of conditional statements added to the benchmark programs by our tool, to make the coverage objectives from the MCC and WM (ABS, AOR, COR and ROR operators) criteria explicit.

Program	**Lines of code**	**Number of added conditionals**			**Program size growth** (1 l.o.c./conditional)		
		MCC	**WM**	**Both**	**MCC**	**WM**	**Both**
base64	324	16	101	117	+5%	+31%	+36%
uniq	700	178	122	300	+25%	+17%	+43%
md5sum	1,021	82	226	308	+8%	+22%	+30%
who	12,311	134	2,037	2,171	+1%	+17%	+18%
TOTAL	**14,356**	**410**	**2,486**	**2,896**	**+3%**	**+17%**	**+20%**
lua	16,633	968	5,278	6,246	+7%	+39%	+46%
php	22,733	2,761	4,756	7,517	+12%	+21%	+33%
libsndfile	51,748	3,454	24,420	27,874	+7%	+47%	+54%
libpng	52,200	5,412	11,451	16,863	+10%	+22%	+32%
libtiff	70,561	6,446	28,525	34,971	+10%	+40%	+50%
libxml2	151,758	89,767	63,478	153,245	+59%	+42%	+101%
sqlite3	171,130	11,247	28,199	39,446	+7%	+16%	+23%
openssl	188,442	15,156	33,097	48,253	+8%	+18%	+26%
TOTAL	**722,205**	**135,211**	**199,204**	**334,415**	**+18%**	**+28%**	**+46%**

Code instrumentation slows down the program builds, as instrumentation itself takes time to complete and the instrumented program is larger to compile. A detailed breakdown of the build slowdown numbers can be found in Table 3 for the MAGMA programs (all build times for the LAVA-M programs are less than one second, so that overhead data are not really meaningful). In total, the instrumentation overhead makes building the MAGMA programs more than seven times slower. It should be noted that most of the slowdown comes from the preprocessing phase, where we leverage the clang-tidy tool to format the original code (e.g. adding braces around single-instruction code blocks in which empty conditionals should be inserted), so that our instrumentation will not break the C syntax. While not a priority in our proof-of-concept, there remain a lot of room to optimise our code instrumentation process, in order to make our tool more usable in the wild. Without code tidying, the build process is "only" more than twice slower.

Table 3. Time overhead caused by instrumentation over the program build process for the Magma benchmark (MCC and WM criteria). Instrumenting a program involves code tidying as a preprocessing step (with clang-tidy), followed by the side-effect extraction phase and the phase where conditionals for the considered labels are inserted in the code. Finally, the resulting code can be built using the make command.

Program	**Build**	**Instrumentation + Build**					**Total overhead**	
	make	**clang-tidy**	**Side-effects**	**Labels**	**make**	TOTAL	with **clang-tidy**	w/o **clang-tidy**
lua	5s	98s	3s	3s	8s	112s	+2,242%	+187%
php	260s	137s	14s	15s	300s	465s	+79%	+26%
libsndfile	48s	342s	47s	46s	62s	497s	+937%	+223%
libpng	5s	100s	2s	3s	17s	122s	+2,245%	+336%
libtiff	25s	368s	7s	9s	48s	431s	+1,657%	+160%
libxml2	21s	1,183s	8s	17s	211s	1,419s	+6,516%	+999%
sqlite3	24s	361s	8s	10s	72s	451s	+1,768%	+272%
openssl	225s	544s	61s	59s	487s	1,150s	+411%	+170%
TOTAL	**613s**	**3,133s**	**150s**	**161s**	**1,205s**	**4,649s**	**+658%**	**+147%**

We sanity check our instrumentation in different ways. We start by we reviewing manually some hand-picked fragments of the programs to verify that there is no visible issue in the instrumented source code and in the resulting compiled binary code. Then, we check that the label-derived conditional statements have not altered the semantics of the programs in two ways. First, we compare manually the behaviour of original and instrumented programs on a curated set of inputs. Second, we check that fuzzing the instrumented programs does not trigger any inexplicable crashes vs. fuzzing the original programs.

6.4 RQ1: impact over fuzzer guidance

Impact on grey-box fuzzing. We run the AFL++ grey-box fuzzer over our sixteen executables from the LAVA-M and MAGMA benchmark, with and without our instrumentation. Tables 4 and 5 then synthesize the observed impact of adding this instrumentation over the fuzzer's guidance. More precisely, the tables detail how the fuzzer's throughput, number of saved seeds, reached (edge) coverage[9] and number of discovered bugs change when the instrumentation is added, as well as a quantification of the uncertainty affecting these data due to the randomness of the fuzzing processes. Interested readers can find even more detailed data in the appendices, including raw throughput and bugs numbers, as well as plots of bugs, seeds and edge coverage evolution across time. In the next paragraphs, we analyse the consolidated results from Tables 4+5 in more details.

The three smallest executables from LAVA-M (base64, uniq and md5sum) are probably too simple to enable a meaningful evaluation of our approach. Indeed, AFL++ quickly saturates over these small programs (see Figure 13) and always detects all of their bugs (with and without our instrumentation, see Table 15 in appendix).

For the remaining thirteen executables (who from LAVA-M and all the MAGMA executables), one can observe that adding our instrumentation always decreases the fuzzer's throughput, whatever the coverage criterion (MCC, WM or both). This effect can be explained by the fact that instrumenting the executable makes it slower to run, as the coverage objectives made explicit must be evaluated every time their corresponding conditional is reached during a run. This mechanically reduces the number of runs that the fuzzer can perform per unit of time. In addition, because of the added

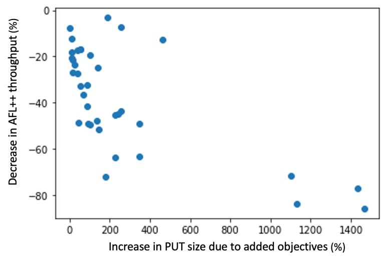
[9]We always measure edge coverage w.r.t. the control-flow graph of the original PUT, i.e. without any instrumentation. When fuzzing instrumented versions of the PUT, we save the generated seeds and then run them over the original PUT, to measure the resulting edge coverage.

Table 4. Impact of making MCC and WM coverage objectives explicit in the PUT's code over AFL++'s average throughput (average of five runs), total number of saved seeds (average of five runs), total number of covered edges (average of five runs) and total number of discovered unique bugs (consolidated over five runs). Reported increase or decrease values are against AFL++ applied on the original PUT.

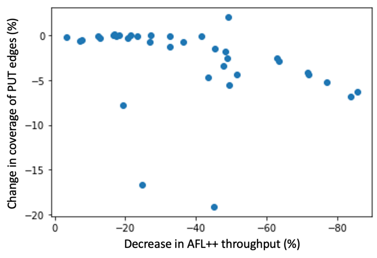
Executable	AFL++ with MCC				AFL++ with WM				AFL++ with MCC + WM			
	Runs/s	Seeds	Edges	Bugs	Runs/s	Seeds	Edges	Bugs	Runs/s	Seeds	Edges	Bugs
base64	+29%	+2%	+2	—	+40%	+1%	—	—	-5%	+2%	+2	—
uniq	+16%	-5%	+7	—	-1%	+12%	+10	—	-21%	+13%	+6	—
md5sum	+18%	-34%	-41	—	+3%	-31%	-41	—	+25%	-24%	-12	—
who	-6%	+19%	+133	+165	-9%	+28%	+6	+98	-19%	+22%	-4	-56
lua	-8%	+6%	-65	—	-33%	+7%	-159	—	-36%	+6%	-99	—
exif	-21%	-19%	-41	+1	-12%	-5%	-13	+1	-27%	-25%	-98	+1
sndfile	-48%	+2%	-239	—	-72%	+39%	-578	—	-64%	+48%	-373	—
libpng_read	-7%	+64%	-33	-1	-3%	+45%	-12	—	-13%	+95%	-16	—
tiff_read_rgba	-49%	-2%	-268	-2	-48%	+11%	-354	-1	-45%	+15%	-158	-1
tiffcp	-49%	-9%	-653	-2	-52%	+18%	-512	-2	-44%	+18%	-543	-2
read_memory	-84%	+35%	-1447	—	-63%	+8%	-556	—	-86%	+53%	-1333	—
xmllint	-72%	+46%	-850	-1	-49%	+12%	+401	-1	-77%	+54%	-1059	-1
sqlite3	-19%	-7%	-2489	—	-25%	-10%	-5297	—	-45%	-19%	-6062	—
server	-17%	-3%	+3	-1	-18%	-5%	-26	-1	-33%	-5%	-47	-1
client	-17%	+2%	+16	—	-27%	—	-20	—	-42%	-1%	-27	—
x509	-18%	+1%	-9	-1	-21%	+1%	-11	—	-24%	+1%	-13	-1

Table 5. P-values from the Mann–Whitney U test [36] corresponding to data reported in Table 4. In a nutshell, each p-value can be seen as an estimated probability that the difference (increase in throughput, reduced number of bugs, etc.) synthesized by the corresponding averaged value in Table 4 would be due to the randomness of the two compared datasets. The p-values in this table are coloured in green if they are smaller than 2% (difference has among the highest probabilities to be real), in orange if they are between 2 and 50% (difference has more probability to be real than caused by randomness) and in red if they are above 50% (difference has less probability to be real than caused by randomness).

Executable	AFL++ with MCC				AFL++ with WM				AFL++ with MCC + WM			
	Runs/s	Seeds	Edges	Bugs	Runs/s	Seeds	Edges	Bugs	Runs/s	Seeds	Edges	Bugs
base64	1	1	0.057	1	0.752	0.752	0.817	1	0.401	0.401	0.057	1
uniq	0.059	0.059	0.058	1	0.011	0.011	0.093	1	0.012	0.012	0.399	1
md5sum	0.012	0.012	0.012	0.723	0.011	0.011	0.012	0.441	0.001	0.001	0.011	0.18
who	0.834	0.834	0.916	0.69	0.059	0.059	0.151	0.151	0.142	0.142	0.916	1
lua	0.69	0.69	0.421	1	0.548	0.548	0.151	0.424	0.841	0.841	0.151	1
exif	0.032	0.032	0.059	0.177	0.841	0.841	1	0.177	0.095	0.095	0.021	0.424
sndfile	0.31	0.031	0.032	1	0.008	0.008	0.008	1	0.008	0.008	0.008	1
libpng_read	0.008	0.008	0.222	0.004	0.008	0.008	0.222	0.424	0.008	0.008	0.053	0.177
tiff_read_rgba	0.69	0.69	0.151	0.18	0.095	0.095	0.095	0.905	0.008	0.008	0.548	0.519
tiffcp	0.016	0.016	0.008	0.017	0.008	0.008	0.016	0.056	0.008	0.008	0.008	0.056
read_memory	0.016	0.016	0.008	0.017	0.008	0.008	0.016	0.056	0.008	0.008	0.008	0.056
xmllint	0.008	0.008	0.008	0.233	0.008	0.008	0.151	0.424	0.008	0.008	0.008	0.233
sqlite3	0.151	0.151	0.016	0.424	0.056	0.056	0.008	0.6	0.008	0.008	0.008	0.233
server	0.841	0.841	0.753	0.424	0.222	0.222	0.222	0.424	0.032	0.032	0.008	0.424
client	0.016	0.016	0.069	1	0.69	0.69	0.042	1	0.095	0.095	0.141	1
x509	0.222	0.222	0.401	0.424	0.69	0.69	0.172	1	0.402	0.402	0.209	0.424



(a) AFL++ throughput decrease vs increase of executable size due to added fine-grained objectives (Magma benchmark).



(b) Change in coverage of executable edges vs AFL++ throughput decrease (Magma benchmark).

Fig. 12. Impact of adding fined-grained objectives over AFL++'s throughput and edge coverage.

conditionals, AFL++'s harness generates more coverage data to be subsequently processed by the fuzzer, slowing down the fuzzing loop. Plotting throughput vs. increase of program size (Figure 12a) shows that the decrease in fuzzer's throughput is well correlated with the relative increase of the PUT's size, caused by the conditionals added by our instrumentation. Put another way, the denser the PUT is instrumented with explicit fine-grained coverage objectives, the more the fuzzer's throughput is usually reduced.

The effect of adding our instrumentation over the number of seeds saved by the fuzzer seems more intricate, as both rises and falls are observed from one setting to another. This complex influence can be mainly explained by a couple of intertwined but opposing forces. Indeed, in the one hand, instrumenting the program with many conditionals reduces the fuzzer's throughput, which means less possibilities to save new seeds in the pool within the same fuzzing budget. In the other hand, as more conditionals are added in the code by our instrumentation, a higher fraction of the generated inputs are considered as interesting by the fuzzer and saved as seeds in the pool. Depending on how these two forces combine during the fuzzer run, the net effect will be either a rise or fall of the number of saved seeds, compared to fuzzing the program without instrumentation.

Adding our instrumentation also impacts the number of PUT's edges covered during a fuzzer run. Most of the time, this impact is negative (i.e. less edges are covered) and quite well correlated with the decrease in fuzzer's throughput (Figure 12b). This can be explained by the fact that, as the fuzzer's throughput decreases, the fuzzer has less chance to discover new branches within the same fuzzing budget. Yet, in some rare occasions, adding our instrumentation leads to an increase, sometimes rather significant, in the number of covered edges. One hypothesis to explain this phenomenon is that some of the additional inputs saved as seeds thanks to our instrumentation revealed key to help the fuzzer unlock the doors of new parts of the PUT, which could not be penetrated when fuzzing without instrumentation.

Finally, our instrumentation has a quite positive impact over the number of unique bugs found in LAVA-M's who executable with the WM criterion, with about one hundred new bugs found and a moderate level of uncertainty due to randomness. On MAGMA, the general trend is that the impact over bug finding is either zero or slightly negative, except for the exif executable, where one additional bug is consistently found, whatever the metric, with a moderate level of uncertainty due to randomness. One way to try and explain this result is by looking at Table 1. One can then notice that who and exif, the two executables where our instrumentation increases the bug finding power, are also the ones which have been seeded with the highest density of bugs among their respective benchmarks. The bug density in who (one bug every six lines of code) is also far higher than in exif (one bug every 1,421 lines of code), which could explain why 98 new bugs were found in who, while only one was found exif. Another way to look at the bug finding results is by focusing on how the number of discovered bugs in who evolves when combining the MCC and WM criteria. Each of the two criteria taken separately enables finding more bugs than without instrumentation (133 with MCC, but with a strong uncertainty, and 98 with WM). Yet, instrumenting the program with the objectives from both criteria together leads to no measurable difference between the number of discovered bugs with and without instrumentation. One hypothesis to explain this behaviour is that, in this last setting, as the fuzzer's throughput gets much more reduced than in the first two, it reduces also the chance for the fuzzer to find bugs within the same fuzzing budget. A reduced fuzzing throughput could also explain why the fuzzer misses some bugs in MAGMA, compared to fuzzing without instrumentation. To sum up, our approach seems better at helping fuzzers clean bug nests (high bug density) than at helping them find a needle in a haystack (low bug density). In addition, as our approach decreases the fuzzer's throughput, it may end up being counterproductive and reduce bug finding power if the throughput reduction is too strong.

Impact on hybrid fuzzing. We also run the QSYM hybrid fuzzer over our sixteen executables from the LAVA-M and MAGMA benchmarks, with and without our instrumentation. We report the results in a similar way to what we did in the previous section for AFL++, with Tables 6 and 7 synthesizing the observed impact of adding this instrumentation over the fuzzer's guidance. The tables details again how the fuzzer's throughput, number of saved seeds, reached (edge) coverage and number of discovered bugs change when the instrumentation is added, as well as the corresponding level of uncertainty. Interested readers can find again even more detailed data in the appendices, including raw throughput and bugs numbers, as well as plots of bugs, seeds and edge coverage evolution across time. In the next paragraphs, we analyse how the consolidated results for AFL++ and QSYM (Tables 4+5 vs 6+7) compare to each other.

As a preliminary remark, by looking at Tables 14 and 15 in the appendices, one can notice that, with or without instrumentation, AFL++ is intrinsically better than QSYM at finding bugs both in LAVA-M and MAGMA. This correlates well with throughput and coverage numbers (Tables 12 and 13, as well as Figures 14 and 17 in appendices), which show that QSYM is twice slower than AFL++ in average and usually ends up covering less edges in the PUT than AFL++ does.

Despite this difference in general performance between the two fuzzers, the impact of our instrumentation over the behaviour of QSYM is similar to the one over AFL++ in many aspects. Indeed, most of the time, we also observe with QSYM a drop in fuzzing throughput and reached edge coverage, as well as a mixed effect on the number of saved seeds. Here again, the number of discovered bugs rises with the who executable and it stays mostly stable or slightly decreases with the MAGMA executables. Yet, in some settings, adding our instrumentation provokes a significant burst in fuzzing throughput, making QSYM more than three times faster than without instrumentation. While QSYM's internal dynamics appear intricate, these bursts seem linked to the fact that our added conditionals result in many over-constrained predicates being produced by symbolic execution. This over-constraining may make the predicates much easier to solve, sometimes freeing a lot of additional CPU time from the constraint solving process for the raw fuzzing process.

Finally, it should be noted that no obvious trend emerges from our results about one criterion (between MCC, WM and the combination of both) being more efficient at guiding the fuzzers to find bugs. Yet, this does not necessarily mean that none of the criteria has better bug-revealing capabilities than the others, but that these capabilities might have been diluted by other effects, like the decrease in fuzzing throughput, which depends both on the picked criterion and the structure of the instrumented code (leading to more or less instrumentation being added to the PUT).

Summary of answer to RQ1 (impact over fuzzer guidance)

Instrumenting the PUT with fine-grained coverage objectives usually leads to a drop in fuzzing throughput and reached edge coverage, as well as a mixed effect on the number of saved seeds. As more objectives are made explicit in the PUT, it gets slower and the fuzzing harness returns more coverage data. This slows down the fuzzing loop and thus lessens the chances of the fuzzer to cover new edges during the same budget. A slower fuzzing loop also means less possibilities to save new seeds within the same budget. However, this effect is counterbalanced as the conditionals added by our instrumentation make a higher fraction of inputs likely to be saved as seeds by the fuzzer. **Our instrumentation can significantly increase bug finding power in the unlikely and hard-to-predict situation where the PUT is seeded with a high density of bugs and where the number of objectives is low enough for not slowing down the fuzzer too much. Yet, most of the time, the impact over bug finding power of fuzzers is either neutral or negative.** While the hybrid fuzzer has more complex internal dynamics and a worse overall performance, the impact of our instrumentation appears similar in many aspects between grey-box and hybrid fuzzing.

6.5 RQ2: impact of carefully selecting coverage objectives

Pruning infeasible/trivial objectives out. We have observed that a high density of fine-grained coverage objectives made explicit in the PUT decreases the fuzzer's throughput and consequently hinders its performance. In this light, the final optional stage of our approach, where infeasible and trivial objectives are pruned out of the PUT before fuzzing, appears as a welcome means to reduce the objective density and recover fuzzing performance.

Table 8 summarises the impact of this pruning stage over the performance of AFL++ with both the MCC and WM objectives over the LAVA-M benchmark. As discussed in the previous section, the results with the three simplest executables should be looked at prudently. In who, removing the infeasible and trivial MCC and WM conditionals cuts the throughput penalty by two, but no measurable improvement in the ability to find bugs was reported. In addition, these changes come at a very high price, as detecting the infeasible and trivial conditionals for LAVA-M required 22 hours and 43 minutes of analysis. This high cost makes our detection approach impossible to scale to the MAGMA programs, as it requires building each program once per inserted conditional, which is intractable in reasonable time for large programs including many conditionals. While using a dedicated static analysis may improve the scalability and performance of our pruning stage, we chose to consider a much simpler way to reduce the objective density in the MAGMA programs. We detail and evaluate this technique in the next subsection.

Discarding subcategories of coverage objectives. A simpler way to reduce the density of fine-grained objectives in instrumented programs is simply discarding some categories of these objectives. One can then hope that the remaining ones will still improve the fuzzer's guidance without impacting too much its throughput.

To test this technique with the MAGMA benchmark, we have notably limited the generation of MCC objectives at three atomic conditions per combination, reducing so the total number of added conditionals by 67%. Table 10 synthesizes the impact of such a discarding of objectives, over the performance of AFL++ with the MAGMA benchmark. In a nutshell, the fuzzer's throughput, as well as the number of covered edges and discovered bugs all tend to increase. Yet, this impact is not

Table 6. Impact of making MCC and WM coverage objectives explicit in the PUT's code over QSYM's average throughput (average of five runs), total number of saved seeds (average of five runs), total number of covered edges (average of five runs) and total number of discovered unique bugs (consolidated over five runs). Reported increase or decrease values are against QSYM applied on the original PUT.

Executable	QSYM with MCC				QSYM with WM				QSYM with MCC + WM			
	Runs/s	Seeds	Edges	Bugs	Runs/s	Seeds	Edges	Bugs	Runs/s	Seeds	Edges	Bugs
base64	+115%	-2%	+6	+1	+52%	-6%	+6	+2	+135%	+1%	+5	-1
uniq	-25%	—	+1	-4	-8%	—	—	-3	+33%	+7%	-22	-2
md5sum	-8%	+3%	+5	+1	+3%	+14%	-13	+1	+10%	+24%	-6	+1
who	+237%	+27%	+12	+2	+5%	+188%	+18	+14	+6%	+235%	+7	+12
lua	+323%	+61%	-168	+1	+218%	-19%	-558	—	+82%	+13%	-498	—
exif	-15%	+21%	+83	—	-11%	+12%	-11	—	-28%	-11%	-60	—
sndfile	-10%	-37%	-928	-1	-30%	+5%	-247	-1	-22%	+16%	-205	—
libpng_read	-92%	+58%	-71	-1	-24%	+43%	-40	-1	+67%	+121%	-12	-1
tiff_read_rgba	-63%	-43%	-1424	-1	-89%	-46%	-1718	-1	-76%	-23%	-1123	-1
tiffcp	-19%	-25%	-1202	—	-52%	-21%	-1282	-1	+1%	-11%	-1614	—
read_memory	+169%	+60%	-406	—	-90%	-4%	-1213	-1	-29%	+52%	-638	-1
xmllint	-57%	+25%	-916	-1	-64%	+15%	-408	-1	-66%	+66%	-518	-1
sqlite3	-40%	+8%	-366	-1	-6%	-20%	-2476	—	-7%	+2%	-1753	-1
server	+3%	+2%	+4	—	-5%	+2%	+27	—	-20%	+3%	+23	—
client	+102%	+2%	-18	—	+5%	+2%	-6	—	+21%	+1%	-26	—
x509	+14%	-1%	-6	+1	+220%	+2%	+15	+1	+6%	+1%	+18	—

Table 7. P-values from the Mann–Whitney U test [36] corresponding to data reported in Table 6. In a nutshell, each p-value can be seen as an estimated probability that the difference (increase in throughput, reduced number of bugs, etc.) synthesized by the corresponding averaged value in Table 6 would be due to the randomness of the two compared datasets. The p-values in this table are coloured in green if they are smaller than 2% (difference has among the highest probabilities to be real), in orange if they are between 2 and 50% (difference has more probability to be real than caused by randomness) and in red if they are above 50% (difference has less probability to be real than caused by randomness).

Executable	QSYM with MCC				QSYM with WM				QSYM with MCC + WM			
	Runs/s	Seeds	Edges	Bugs	Runs/s	Seeds	Edges	Bugs	Runs/s	Seeds	Edges	Bugs
base64	0.6	0.6	0.011	0.742	0.094	0.094	0.014	0.443	1	1	0.014	0.373
uniq	0.75	0.75	1	0.11	0.672	0.672	1	0.16	0.036	0.036	0.841	0.914
md5sum	0.199	0.199	0.396	0.286	0.02	0.02	0.672	0.911	0.011	0.011	0.675	0.386
who	0.016	0.016	0.402	0.059	0.008	0.008	0.346	0.012	0.008	0.008	0.841	0.016
lua	0.008	0.008	0.295	0.177	0.008	0.008	0.011	1	0.151	0.151	0.012	1
exif	0.008	0.008	0.008	1	0.008	0.008	0.548	1	0.917	0.917	0.249	1
sndfile	0.012	0.012	0.008	0.083	0.094	0.094	0.008	0.403	0.012	0.012	0.008	0.434
libpng_read	0.008	0.008	0.016	0.424	0.008	0.008	1	0.424	0.008	0.008	0.548	0.424
tiff_read_rgba	0.008	0.008	0.008	0.027	0.012	0.012	0.008	0.015	0.008	0.008	0.008	0.015
tiffcp	0.008	0.008	0.008	0.041	0.142	0.142	0.008	0.041	0.151	0.151	0.008	0.403
read_memory	0.008	0.008	0.056	0.177	0.151	0.151	0.008	0.067	0.008	0.008	0.008	0.067
xmllint	0.222	0.222	0.095	0.004	0.032	0.032	0.222	0.004	0.008	0.008	0.032	0.004
sqlite3	0.031	0.31	0.222	0.146	1	1	0.031	0.319	0.548	0.548	0.31	0.07
server	0.059	0.059	0.599	0.27	0.142	0.142	0.036	0.631	0.021	0.021	0.248	0.27
client	0.021	0.0221	0.012	1	0.021	0.021	0.523	1	0.094	0.094	0.012	1
x509	0.6	0.6	0.832	0.424	0.151	0.151	0.598	0.424	0.568	0.548	0.399	1

Table 8. Impact of pruning infeasible and trivial MCC and WM coverage objectives over AFL++'s average throughput (average of five runs), total number of saved seeds (average of five runs), total number of covered edges (average of five runs) and total number of discovered unique bugs (consolidated over five runs). Reported increase or decrease values are against AFL++ applied on the original PUT.

Executable	AFL++ with MCC+WM					AFL++ with pruned MCC+WM				
	Labels	Runs/s	Seeds	Edges	Bugs	Labels	Runs/s	Seeds	Edges	Bugs
base64	117	-5%	+2%	+2	—	62	+1%	-2%	+2	—
uniq	300	-21%	+13%	+6	—	165	-12%	-3%	+5	—
md5sum	308	+25%	-24%	-12	—	159	+8%	-10%	-1	—
who	2,171	-19%	+22%	-4	-56	150	-10%	+18%	+3	+96

Table 9. P-values from the Mann–Whitney U test [36] corresponding to data reported in Table 8. In a nutshell, each p-value can be seen as an estimated probability that the difference (increase in throughput, reduced number of bugs, etc.) synthesized by the corresponding averaged value in Table 8 would be due to the randomness of the two compared datasets. The p-values in this table are coloured in green if they are smaller than 2% (difference has among the highest probabilities to be real), in orange if they are between 2 and 50% (difference has more probability to be real than caused by randomness) and in red if they are above 50% (difference has less probability to be real than caused by randomness).

Executable	AFL++ with MCC+WM				AFL++ with pruned MCC+WM			
	Runs/s	Seeds	Edges	Bugs	Runs/s	Seeds	Edges	Bugs
base64	0.401	0.401	0.057	1	0.917	0.917	0.743	0.424
uniq	0.012	0.012	0.399	1	0.53	0.53	0.841	0.1
md5sum	0.01	0.01	0.011	0.18	0.173	0.173	0.841	1
who	0.142	0.142	0.916	1	0.008	0.008	0.344	1

strong enough to reverse the tide, as the overall trend is still not as good as using AFL++ without any instrumentation.

Summary of answer to RQ2 (impact of careful objective selection)

Pruning infeasible/trivial objectives out before fuzzing seems capable of significantly reducing fuzzer slowdown. Yet, **more work is needed to make objective pruning fast enough and scalable** in practice. A simpler way to reduce the density of fine-grained objectives in instrumented programs is discarding some categories of these objectives. Our trials with large programs containing a low density of bugs show that this approach can improve the fuzzing performance, but without making the fuzzer gain the ability to find more bugs than without instrumentation.

Table 10. Impact of limiting MCC objectives at three atomic conditions per combination over AFL++'s average throughput (average of five runs), total number of saved seeds (average of five runs), total number of covered edges (average of five runs) and total number of discovered unique bugs (consolidated over five runs). Reported increase or decrease values are against AFL++ applied on the original PUT.

Executable	AFL++ with MCC					AFL++ with 3-bounded MCC				
	Labels	Runs/s	Seeds	Edges	Bugs	Labels	Runs/s	Seeds	Edges	Bugs
lua	968	-8%	+6%	-65	—	952	-5%	+4%	-100	—
exif	2,761	-21%	-19%	-41	+1	1,481	+2%	+4%	+10	+1
sndfile	3,454	-48%	+2%	-239	—	2,686	+8%	+3%	-151	—
libpng_read	5,412	-7%	+64%	-33	-1	1652	+14%	+9%	+26	—
tiff_read_rgba	6,446	-49%	-2%	-268	-2	3,758	-16%	+7%	-89	-1
tiffcp		-49%	-9%	-653	-2		-6%	+1%	-446	-2
read_memory	89,767	-84%	+35%	-1447	—	19,959	-20%	-4%	-197	—
xmllint		-72%	+46%	-850	-1		-16%	+5%	-122	-1
sqlite3	11,247	-19%	-7%	-2489	—	7,071	-9%	-11%	-1854	—
server	15,156	-17%	-3%	+3	-1	6,516	+10%	-1%	+16	—
client		-17%	+2%	+16	—		+9%	+2%	-3	—
x509		-18%	+1%	-9	-1		+31%	+3%	+3	-1

Table 11. P-values from the Mann–Whitney U test [36] corresponding to data reported in Table 10. In a nutshell, each p-value can be seen as an estimated probability that the difference (increase in throughput, reduced number of bugs, etc.) synthesized by the corresponding averaged value in Table 10 would be due to the randomness of the two compared datasets. The p-values in this table are coloured in green if they are smaller than 2% (difference has among the highest probabilities to be real), in orange if they are between 2 and 50% (difference has more probability to be real than caused by randomness) and in red if they are above 50% (difference has less probability to be real than caused by randomness).

Executable	AFL++ with MCC				AFL++ with 3-bounded MCC			
	Runs/s	Seeds	Edges	Bugs	Runs/s	Seeds	Edges	Bugs
lua	0.69	0.69	0.421	1	0.69	0.69	0.31	1
exif	0.032	0.032	0.059	0.177	0.346	0.346	0.209	0.424
sndfile	0.31	0.31	0.032	1	0.222	0.222	0.056	1
libpng_read	0.008	0.008	0.222	0.004	0.094	0.094	0.548	0.177
tiff_read_rgba	0.69	0.69	0.151	0.18	0.032	0.032	0.548	0.905
tiffcp	0.016	0.016	0.008	0.017	0.421	0.421	0.008	0.056
read_memory	0.008	0.008	0.008	0.02	0.151	0.151	0.016	1
xmllint	0.008	0.008	0.008	0.233	0.016	0.016	0.075	0.424
sqlite3	0.151	0.151	0.016	0.424	0.056	0.056	0.016	1
server	0.841	0.841	0.753	0.424	1	1	0.209	1
client	0.016	0.016	0.69	1	0.008	0.008	0.249	1
x509	0.222	0.222	0.401	0.424	0.032	0.032	0.675	0.424

6.6 Discussion and future work

Summary of performed research. While software testing researchers have defined standard coverage metrics that are finer-grained than branch coverage, these are not used in state-of-the-art fuzzers, which rely on branch coverage or ad hoc mechanisms [10–12] for guidance. In this work, we have challenged this situation. First, we have developed a mechanism for existing fuzzers to support finer-grained coverage objectives out-of-the-box, by making these objectives explicit as new branches in the PUT code, while carefully avoiding any impact on the PUT's semantics. Second, we have taken advantage of this mechanism to evaluate the impact of guiding state-of-the-art fuzzers (namely AFL++ and QSYM) with state-of-the-art fine-grained coverage criteria (namely Multiple Conditions Coverage and Weak Mutation Coverage) over the fuzzing performance.

Main conclusions. The experiments that we have performed for this paper have convinced us that our mechanism to support additional testing objectives, by adding new branches in the PUT, seemed effective. We have used it here to encode objectives from fine-grained coverage criteria, but we believe that it has the potential to become a *lingua franca* for providing fuzzers with additional guidance at runtime. It could notably be used to encode human directives about interesting program states or to make explicit bug preconditions automatically computed by static analysers.

We report an essentially negative impact of guiding fuzzers with fine-grained coverage criteria. The fine-grained objectives added by our approach had indeed either a zero or a negative profitability in most of the studied situations. The profitability of a set of fine-grained objectives is the difference between the gain (or penalty) provided by keeping the seeds that satisfy these objectives and the overhead caused by the need to monitor and react to their coverage. A positive profitability leads to an increase in fuzzing performance, while a negative profitability decreases it. Our results indicate that two of the factors that seem likely to increase profitability are a high bug density coupled with a low objective density:

- A *high bug density* improves the gain brought by fine-grained objectives compared to a low bug density, probably because of the very nature of the guidance provided by fine-grained criteria. Fine-grained objectives indeed enable a much denser sampling of the subtle local differences of behaviour in the code, probably at the expense of a broader coverage of the complete codebase;
- A *low objective density* reduces the overhead due to fine-grained objectives by limiting the induced decrease in fuzzing throughput, compared to a high objective density. Objective density is determined by the chosen coverage criterion and the particular structure of the PUT code.

As it appears difficult to predict the profitability of a set of fine-grained objectives a priori, before actually fuzzing the instrumented PUT, and given that positive profitability appears infrequent in practice, we do not recommend fine-grained coverage criteria as a general means to guide fuzzers.

Future work. We think that additional research would be useful to check whether leveraging fine-grained criteria could be beneficial in some possibly favorable use cases:

- A first favorable use case could be not to instrument the whole codebase with some explicit fine-grained objectives, but only small parts of it, like those that are the most sensitive or that should be the most fragile (e.g. because they have been coded or patched recently). Such an *à la carte* instrumentation could avoid penalising the fuzzing speed too much by adding only a limited number of objectives (low overhead), while still enabling a denser sampling of sensitive or fragile code behaviours (high gain), resulting possibly in a profitable instrumentation with fine-grained objectives;

- A second favorable use case could be running first a classical fuzzing campaign, relying simply on branch coverage, to explore the code as much as possible at full speed, and then, if necessary, using the collected seeds for a subsequent finer-grained fuzzing campaign, to stress more carefully all the discovered branches in the code.

It also remains an open question whether and how much it would be beneficial (1) to use an improved static pre-identification of polluting [30] and low-profit fine-grained objectives, (2) to remove the covered fine-grained objectives on-the-fly through binary patching [37], and (3) to abandon the ability to reuse existing fuzzers out-of-the-box, in order to enable a tighter integration between test objectives generically encoded with labels and fuzzing algorithms. Providing evidence of the actual interest (or lack of interest) of these three research trails are other interesting directions for future work.

6.7 Threats to validity

Threats to internal validity. A first class of threats to the internal validity of our conclusions arise because the software artifacts that we used, including (1) our toolset to instrument code with explicit fine-grained coverage objectives, (2) the studied AFL++ and QSYM fuzzers and (3) our experimental infrastructure to run fuzzing campaigns and collect relevant data, could be defective. However, we have crosschecked our results in several ways. First, some hand-picked fragments of the instrumented programs have been manually inspected and verified, in order to make sure that the code instrumentation process did produce the expected source code and that the instrumentation was not tampered by compilation to assembly code. Second, the behaviour of the original and instrumented programs were manually compared on a curated set of inputs, to make sure that the inserted label-derived conditional statements had not altered the semantics of the programs. To double-check the proper conservation of program semantics modulo our instrumentation, we have also taken advantage of the performed massive fuzzing campaigns, to make sure that the instrumented code did not trigger any inexplicable crashes compared the original one. Our experimental infrastructure has been tested on small-scale fuzzing campaigns, where we have checked that it was producing consistent results for a set of significantly different fuzzer configurations. All these sanity checks succeeded. In addition, the code of our instrumentation toolset and experimental infrastructure has been reviewed by at least two and up to three different developers. It has been made available as an open-source artefact for review by the community. At the heart of our experimental evaluation, AFL++ is a solid, community-maintained and open-source fuzzer. By September 2022, the project had been starred more than three thousand times on Github and it was reported to have led to the discovery of dozens of CVEs in various applications. As an additional fuzzer to support our evaluation, QSYM is probably closer to a research prototype than an all-weather tool like AFL++. Still, it has been presented at the prestigious USENIX Security conference, where it won a distinguished paper award. It has also been open-sourced and starred more than five hundreds times on GitHub, by September 2022. Finally, it is reported to have led to the discovery of 13 previously unknown vulnerabilities in eight non-trivial programs, including ffmpeg and OpenJPEG.

A second class of threats to the internal validity of our conclusions is that our fuzzing performance results might not be significant, due to the highly random nature of the fuzzing processes. However, we have mitigated this threat by averaging all the collected numbers over five different twenty-four hours runs of the fuzzer, for a total two and a half years of CPU time dedicated to our experiments. In addition, we have systematically estimated and reported the uncertainty in our results using a state-of-the-art statistical hypothesis test.

Threats to external validity. Common to all empirical studies, this one may be of limited generalisability. To reduce this threat, we have performed our experiments over twelve various open-source programs for a total of more than seven hundreds of thousands of lines of code. These programs exhibit a wide range of sizes, different coding styles (leading to different profiles of fine-grained coverage objectives) and belong to two state-of-the-art fuzzing benchmarks, involving either high-density artificial bugs or low-density real bugs. To perform our experiments, we have considered two state-of-the-art coverage criteria, embodying two of the most common approaches to derive fine-grained coverage objectives, based either on analysing logical expressions at decision points or on simulating common faults with mutations operators. The experiments were performed with two both recent and popular state-of-the-art fuzzers, which implement cutting edge capabilities either in grey-box or hybrid fuzzing.

7 RELATED WORK

Several works have considered providing better guidance to fuzzers for selecting seeds that trigger interesting program behaviours.

Aschermann et al. [38] 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. It would also be an interesting future work direction to investigate whether labels could become a *lingua franca* for user annotations to guide fuzzers.

Wang et al. [39] 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 needing more 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 [40] proposed AFL-HIER: a 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 either on shallow standard metrics, like block and branch coverage, or on their own ad-hoc variants of branch coverage, while our work proposes to leverage the many fine-grained metrics widely studied in the software testing literature. However, our method does also increase the number of inputs taken as seeds and thus hampers the performance of the fuzzer for this reason as well. Taking inspiration from the multi-level approach proposed by Wang et al. to cluster seeds and tame the performance reduction in our approach is a promising lead for future work.

Ankou [41] 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 an exhaustive metric delivers far too much data, resulting in seed explosion (similarly to the path explosion occurring in symbolic execution, which is also based on path coverage). 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. [42] 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.

A blog post from 2016 [43] 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.

At the same time or following the initial publication of this pre-registered paper at the FUZZING'22 workshop, six groups of authors [19–21, 44–46] have disclosed their efforts to look at a possible combination between fuzzing and either mutation or data-flow coverage, which we discuss in the forthcoming paragraphs. This signals an emerging trend where the fuzzing community try and take advantage of the existing body of work on fine-grained code-coverage criteria.

Quian et al. [19] run the inputs generated by the fuzzer against a set of mutants of the PUT. Inputs that (strongly) kill these mutants are given more chance to be selected for further mutation by (a modified version of) the fuzzer. In the one hand, this paper also uses a fine-grained coverage metric (together with edge coverage) to guide a fuzzer and the considered metric (strong mutations) is out of reach of our approach (but we still handle a close and less costly approximation of it: weak mutations). In the other hand, contrary to this work, our approach can handle a wide set of various metrics generically and there is no need for any modification of the fuzzer. It is interesting to notice that the issues faced by the authors of this work and the results that they have obtained can be related to ours. First, while the used mutation operators produced up to dozens of thousands of mutants per PUT, only ten mutants had to be considered during each fuzzing campaign to "ensure the efficiency of fuzzing". Second, the impact on fuzzing performance, measured in terms of branch and mutation coverage, reveals that strong mutations do not provide a significant and unconditional advantage against branch coverage alone.

Mantovani et al. [20] modify the instrumentation mechanism of the AFL++ fuzzer to reward inputs covering new variable definition-use pairs, in addition to control edges. In the one hand, this paper also uses a fine-grained coverage metric (together with edge coverage) to guide a fuzzer, and, while the considered metric (a variant of data-flow coverage) is conceptually supported by our approach (see encoding of data-flow test objectives with labels in Figure 5), this support has not been evaluated in practice. In the other hand, contrary to this work, our approach handles a wide set of various metrics generically without modifying the fuzzer. As in our discussion of Quian et al. in the previous paragraph, it is interesting to notice that the issues and results reported by Mantovani et al. can also be related to ours. First, solutions had also to be found to reduce the number of considered additional test objectives, in order to avoid overwhelming the fuzzer and to reduce the overhead. These solutions involve considering the definition-use pairs only from a subset of the PUT's variables, as well as pruning out redundant objectives [30]. Second, the authors also report that their technique slows down the fuzzer (by 10-14% in average) and reduces the total number of edges that it covers. In addition, they report that their technique can discover different bugs, compared to the original fuzzer, on code with a "data-dependent structure". Yet, such code also yields a higher density of additional test objectives, which could penalise the fuzzer by "injecting too much instrumentation".

Concomitantly to the initial publication of this work, two other preliminary reports [21, 45] about fuzzing and fine-grained coverage criteria were also presented at the FUZZING'22 workshop. Herrera et al. [21] provide a preliminary evaluation of a data-flow-guided fuzzer. The described early results suggest that (1) pruning strategies (like considering the definition-use pairs only from a subset of the PUT's variables) are required to avoid reducing the fuzzing throughput too much and (2) edge coverage remains a better metric than data-flow coverage in general, but shows promises with specific sorts of code. Groce et al. [45] propose to spend half of the fuzzing budget running the fuzzer on mutants of the PUT, and then to use the produced seeds as initial seeds to fuzz the PUT itself. The preliminary results suggest that (1) fuzzing each mutant for five minutes within a 10 hours total fuzzing budget implied that only a tiny subset of the mutants (1-2%) could actually be fuzzed and (2) the approach seems to improve the bug finding power of the fuzzer on the *fuzzgoat* bug-laden program (about 1k lines of code).

Finally, two (non-peer-reviewed) research proposals have recently been made public about combining mutation coverage and fuzzing. Laybourn et al. [44] propose to use mutation coverage to augment the coverage feedback of fuzzers, while Gopinath et al. [46] evaluate the challenges of using mutation coverage to build fuzzing benchmarks.

8 CONCLUSION

In this work, we have taken advantage of the large body of research over fine-grained code coverage criteria and used these criteria as additional 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. In addition, all the fuzzing mechanisms designed to penetrate hard-to-cover branches will serve to help covering the additional metrics objectives. We have used this approach to evaluate the impact of supporting two representative fine-grained coverage metrics (multiple condition coverage and weak mutation) over the performance of a state-of-the-art grey-box and hybrid fuzzer (AFL++ and QSYM) with the standard LAVA-M and MAGMA benchmarks. This evaluation has revealed that our guidance mechanism, where the fuzzed code is instrumented with additional branches, is effective and could possibly be leveraged in other contexts, like with human directives or bug preconditions from static analysers. However, this evaluation has also revealed that the impact of fine-grained metrics was hard to predict before fuzzing and most of the time either neutral or negative, so that we do not recommend using them as a general means to guide fuzzers. Yet, it remains an open question whether such metrics could be useful in some specific favorable circumstances, like for limited parts of the codebase or as a complement to classical fuzzing campaigns. Overall, as the interest in fine-grained coverage metrics is rising in the fuzzing community, we provide in this work a significant step towards better understanding how these metrics could or could not be useful in the context of fuzzers.

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 *Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. Association for Computing Machinery, 2017.

[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] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[7] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08.

[8] 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.

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

[10] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "Fuzzfactory: Domain-specific fuzzing with waypoints," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360600

[11] N. Coppik, O. Schwahn, and N. Suri, "Memfuzz: Using memory accesses to guide fuzzing," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2019, pp. 48–58. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST.2019.00015

[12] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20. USA: USENIX Association, 2020.

[13] 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.

[14] A. Parsai and S. Demeyer, "Comparing mutation coverage against branch coverage in an industrial setting," *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 4, p. 365–388, aug 2020. [Online]. Available: https://doi.org/10.1007/s10009-020-00567-y

[15] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," *Department of Information and Software Systems Engineering, George Mason University, Tech. Rep. ISSE-TR-96-100*, 1996.

[16] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[17] 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.

[18] 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

[19] R. Qian, Q. Zhang, C. Fang, and L. Guo, "Investigating coverage guided fuzzing with mutation testing," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, ser. Internetware '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 272–281. [Online]. Available: https://doi.org/10.1145/3545258.3545285

[20] A. Mantovani, A. Fioraldi, and D. Balzarotti, "Fuzzing with data dependency information," in *EuroS&P 2022, 7th IEEE European Symposium on Security and Privacy, 6-10 June 2022, Genoa, Italy*, IEEE, Ed., Genoa, 2022.

[21] A. Herrera, M. Payer, and A. L. Hosking, "Registered report: Dataflow."

[22] 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.

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

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] A. Offutt and S. Lee, "An empirical evaluation of weak mutation," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 337–344, 1994.

[30] 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

[31] 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.

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

[33] 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.

[34] 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.*

[35] 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).*

[36] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1–10. [Online]. Available: https://doi.org/10.1145/1985793.1985795

[37] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21).* USENIX Association, Aug. 2021, pp. 1683–1700. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy

[38] 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.

[39] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *RAID 2019.* USENIX Association.

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

[41] 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. ACM, 2020.

[42] 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)*, 2021.

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

[44] I. Laybourn, V. Vikram, R. Sanna, A. Li, and R. Padhye, "Guiding greybox fuzzing with mutation testing," 2022.

[45] A. Groce, G. T. Kalburgi, C. Le Goues, K. Jain, and R. Gopinath, "Registered report: First, fuzz the mutants."

[46] R. Gopinath, P. Görz, and A. Groce, "Mutation analysis: Answering the fuzzing challenge," 2022. [Online]. Available: https://arxiv.org/abs/2201.11303

ACKNOWLEDGMENTS

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, from the France FUI CAESAR project and from the Chateaubriand Fellowship Program of the Embassy of France in the United States.

DETAILED FUZZING DATA FOR AFL++

This appendix gathers detailed data about the measured impact of making MCC and WM coverage objectives explicit in the PUT's code over AFL++'s behaviour. More precisely, it details the observed impact over the fuzzer's throughput (Table 12), the number of seeds saved by the fuzzer (Figure 13), the reached edge coverage (Figure 14) and the number of discovered bugs (Table **??** and Figure 15).

Table 12. Impact of making MCC and WM coverage objectives explicit in the code over AFL++'s throughput (average of five runs).

Executable	**Average fuzzer throughput** (PUT executions/second)			
	AFL++ (baseline)	with **MCC objectives**	with **WM objectives**	with **MCC + WM objectives**
base64	1.3K	1.7K (+29%)	1.8K (+39%)	1.3K (-)
uniq	1.7K	2.0K (+15%)	1.7K (-1%)	1.4K (-21%)
md5sum	659	777 (+18%)	681 (+3%)	824 (+25%)
who	1.2K	1.1K (-6%)	1.1k (-9%)	951 (-18%)
AVERAGE	**1.2K**	**1.4K (+14%)**	**1.3K (+8%)**	**1.1K (-8%)**
lua	746	689 (-8%)	502 (-33%)	475 (-36%)
exif	173	137 (-21%)	152 (-12%)	127 (-27%)
sndfile	916	472 (-48%)	256 (-72%)	334 (-64%)
libpng_read	1.2K	1.1K (-7%)	1.2K (-3%)	1.0K (-13%)
tiff_read_rgba	1.6K	832 (-49%)	851 (-48%)	894 (-45%)
tiffcp	1.3K	664 (-49%)	633 (-52%)	740 (-44%)
read_memory	977	159 (-84%)	361 (-63%)	140 (-86%)
xmllint	792	224 (-72%)	403 (-49%)	183 (-77%)
sqlite3	705	569 (-19%)	531 (-25%)	388 (-45%)
server	46.8	38 (-17%)	39 (-17%)	32 (-32%)
client	95	79 (-17%)	69 (-27%)	56 (-42%)
x509	446	365 (-18%)	350 (-21%)	341 (-23.5%)
AVERAGE	**750**	**444 (-41%)**	**446 (-41%)**	**392 (-48%)**

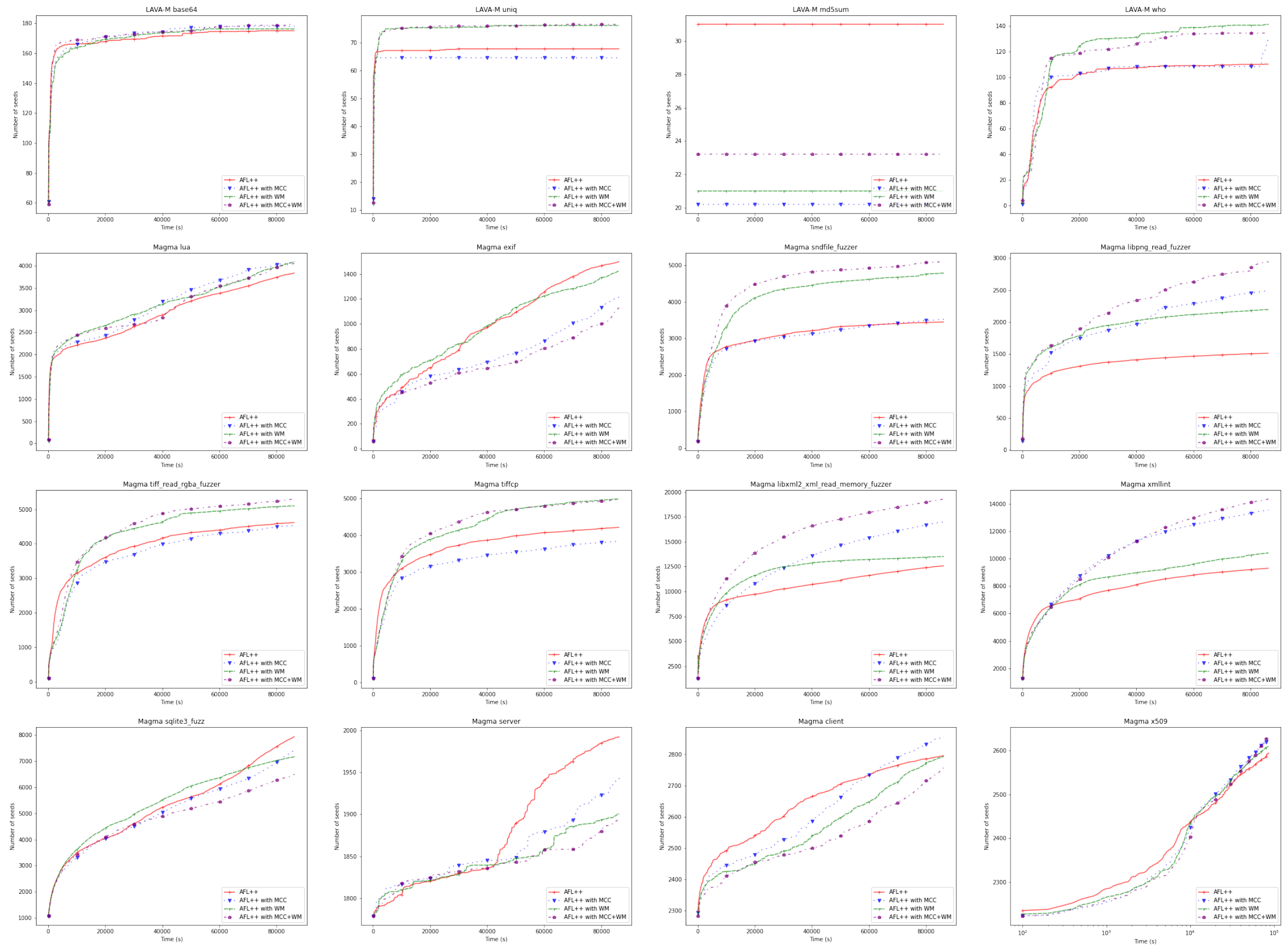


Fig. 13. Impact of making MCC and WM coverage objectives explicit in the code over AFL++'s seed pool (average of five runs).

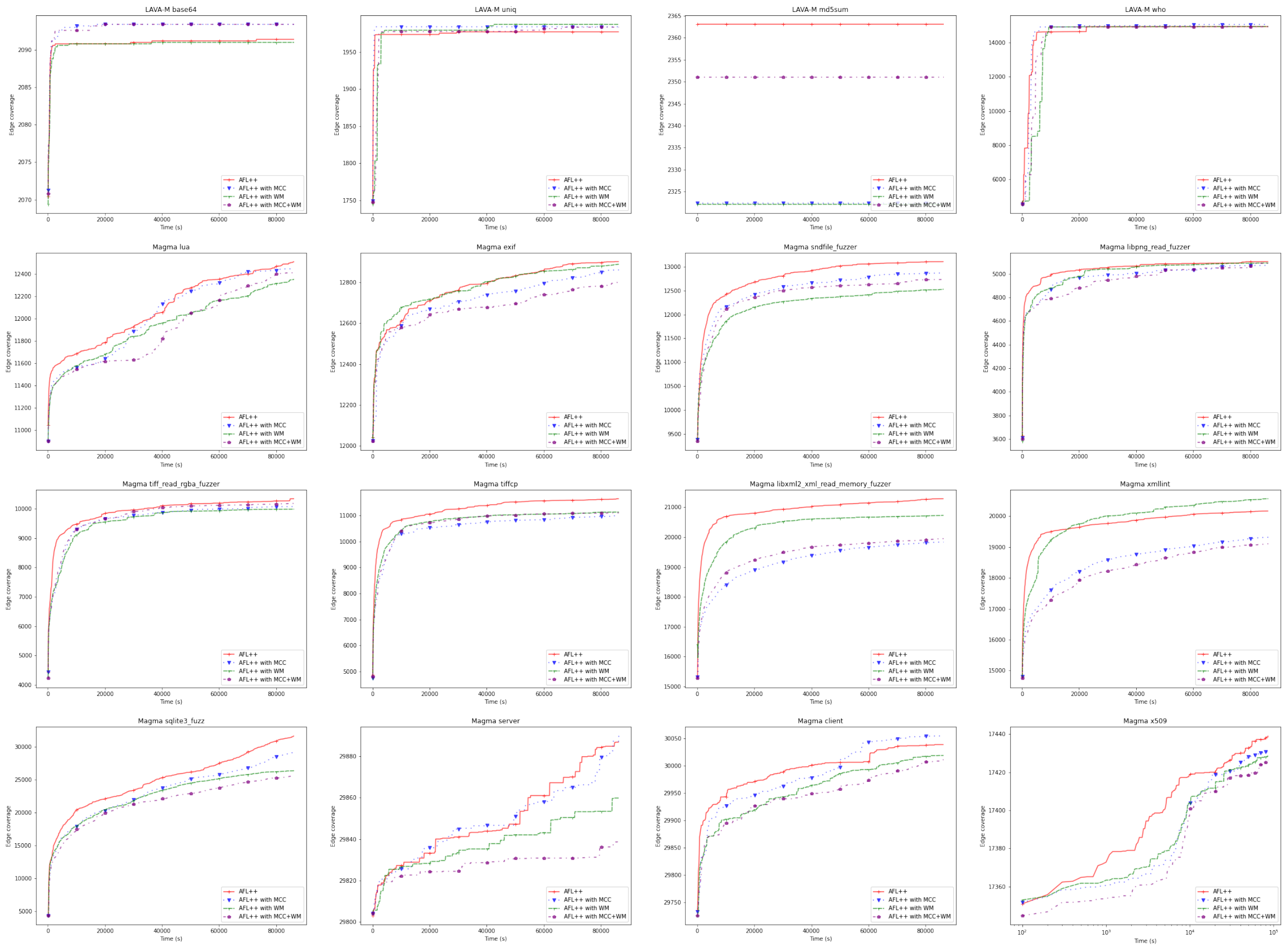


Fig. 14. Impact of making MCC and WM coverage objectives explicit in the code over the edge coverage reached by AFL++ (average of five runs).

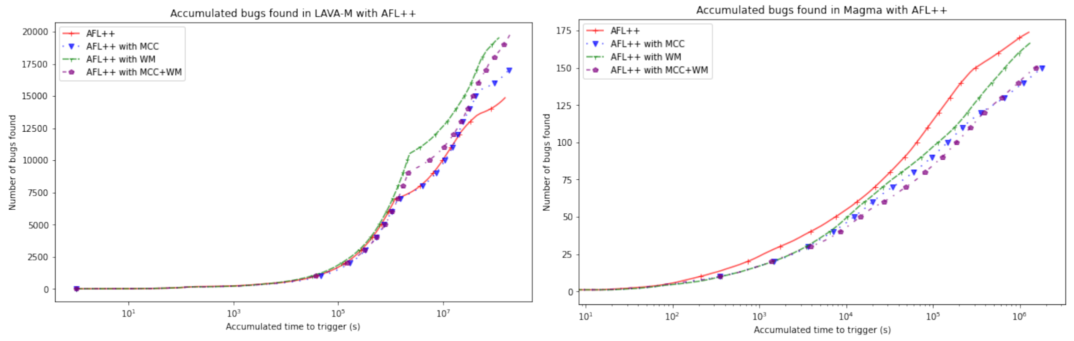


Fig. 15. Impact of making MCC and WM coverage objectives explicit in the code over the time to trigger bugs with AFL++ (consolidated over five runs and the whole LAVA-M and MAGMA benchmarks).

DETAILED FUZZING DATA FOR QSYM

This appendix gathers detailed data about the measured impact of making MCC and WM coverage objectives explicit in the PUT's code over QSYM's behaviour. More precisely, it details the observed impact over the fuzzer's throughput (Table 13), the number of seeds saved by the fuzzer (Figure 16), the reached edge coverage (Figure 17) and the number of discovered bugs (Table **??** and Figure 18).

Table 13. Impact of making MCC and WM coverage objectives explicit in the code over QSYM's throughput (average of five runs).

Program	**Average fuzzer throughput** (executions/second)			
	QSYM (baseline)	with **MCC objectives**	with **WM objectives**	with **MCC + WM objectives**
base64	224	481 (+115%)	341 (+52%)	527 (+135%)
uniq	1.7K	1.3K (-25%)	1.6K (-8%)	2.3K (+33%)
md5sum	63	58 (-8%)	65 (+3%)	69 (+10%)
who	66	221 (+237%)	69 (+5%)	69 (+6%)
AVERAGE	**513**	**515** (-)	**518 (+1%)**	**741 (+45%)**
lua	50	213 (+323%)	160 (+218%)	92 (+82%)
exif	55	47 (-15%)	49(-11%)	40 (-28%)
sndfile	62	56 (-9%)	44 (-30%)	49 (-22%)
libpng_read	1.6K	129 (-92%)	1.3K (-24%)	2.7K (+67%)
tiff_read_rgba	854	316 (-63%)	94 (-89%)	205 (-76%)
tiffcp	64	51 (-19%)	30 (-52%)	64 (+1%)
read_memory	252	677 (+168%)	26 (-90%)	179 (-29%)
xmllint	62	27 (-57%)	23 (-64%)	21 (-66%)
sqlite3	688	417 (-39%)	647 (-6%)	639 (-7%)
server	24	25 (-5%)	24 (-5%)	20 (-20%)
client	70	141 (+102%)	73 (+5%)	85 (+21%)
x509	167	190 (+14%)	535 (+220%)	176 (+6%)
AVERAGE	**329**	**191 (-42%)**	**250 (-24%)**	**356 (+8%)**

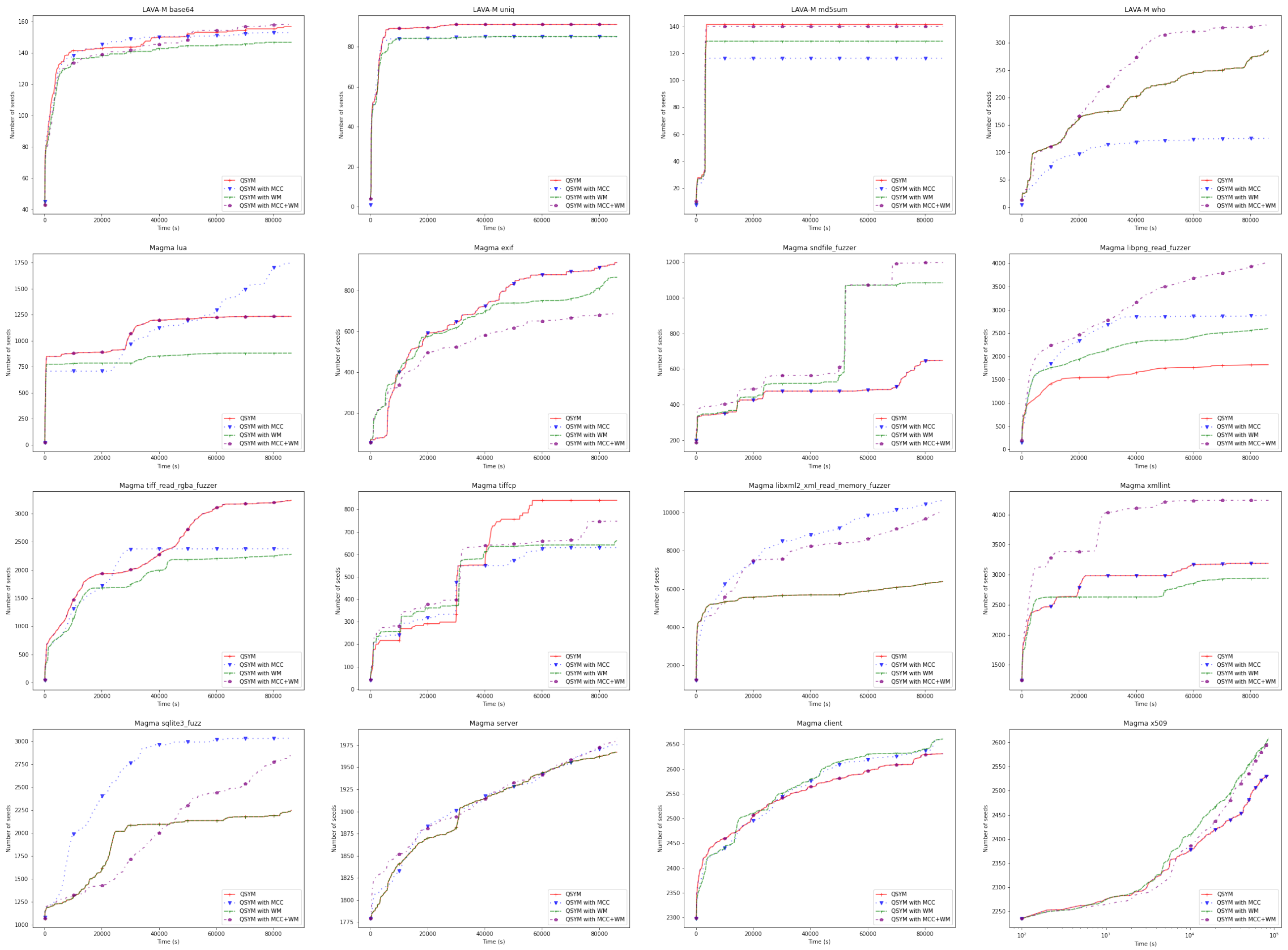


Fig. 16. Impact of making MCC and WM coverage objectives explicit in the code over QSYM's seed pool (average of five runs).

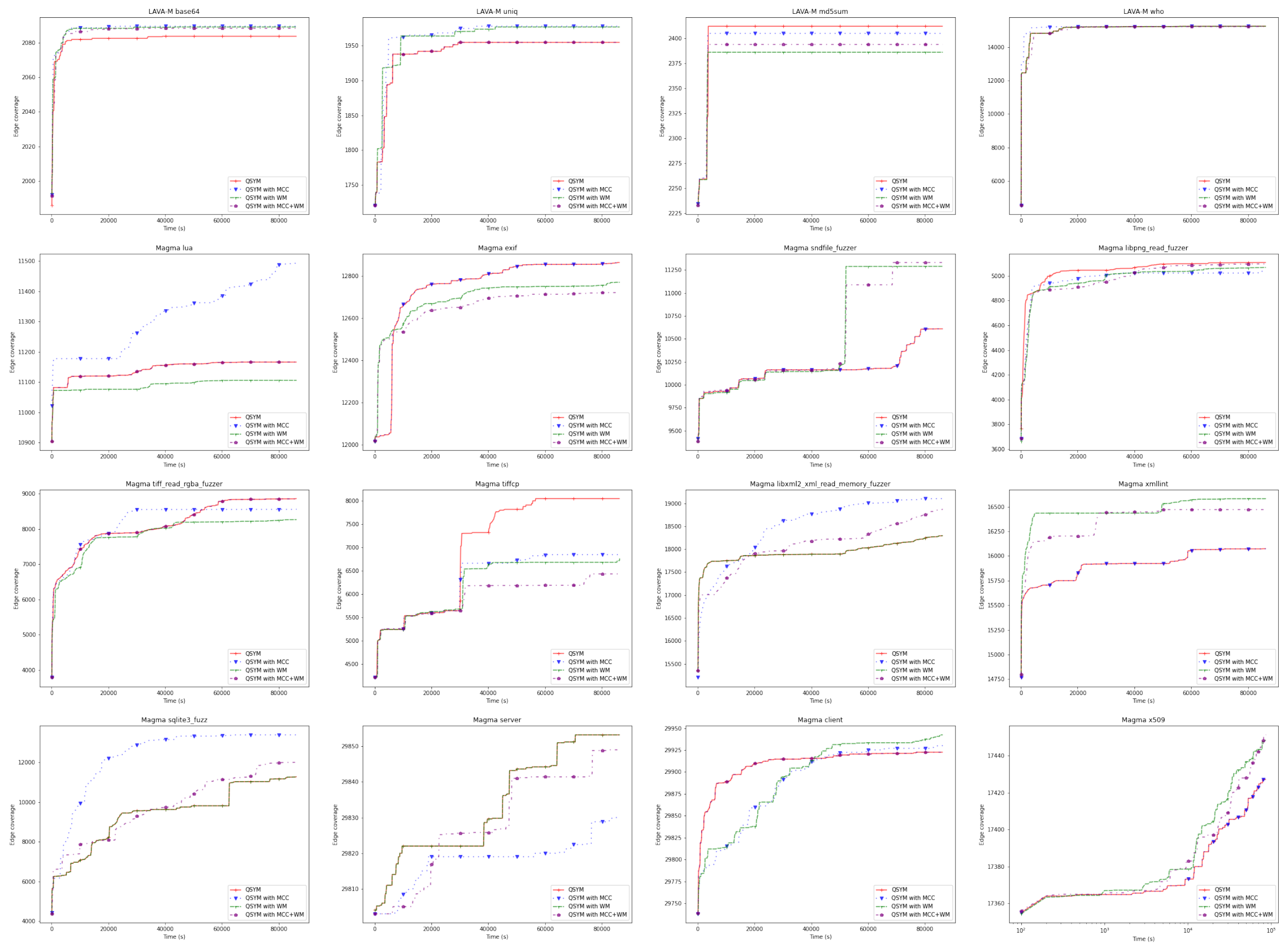


Fig. 17. Impact of making MCC and WM coverage objectives explicit in the code over the edge coverage reached by QSYM (average of five runs).

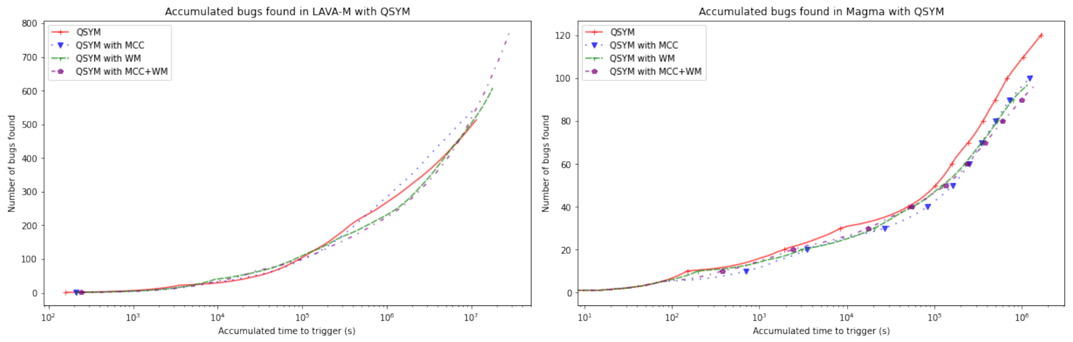


Fig. 18. Impact of making MCC and WM coverage objectives explicit in the code over the time to trigger bugs with QSYM (consolidated over five runs and the whole LAVA-M and MAGMA benchmarks).

DETAILED UNIQUE BUG FINDING DATA FOR AFL++ AND QSYM

Table 14. Impact of making MCC and WM coverage objectives explicit in the code over the total number of unique bugs found by AFL++ over five runs. Variance between the five runs is reported through the measured interquartile range (IQR) and standard deviation (SD).

Program	Bugs	AFL++			AFL++ with MCC			AFL++ with WM			AFL++ with MCC + WM		
		Found bugs	IQR	SD	Found bugs	IQR	SD	Found bugs	IQR	SD	Found bugs	IQR	SD
base64	48	48 (100%)	0	0	48 (100%)	0	0	48 (100%)	0	0	48 (100%)	0	0
uniq	29	29 (100%)	0	0	29 (100%)	0	0	29 (100%)	0	0	29 (100%)	0	0
md5sum	57	57 (100%)	2	1.41	57 (100%)	1	0	57 (100%)	0	0	57 (100%)	0	0
who	2234	1412 (63%)	71	57.92	1577 (71%)	479	249.82	1510 (68%)	137	80.52	1356 (61%)	16	67.55
TOTAL	**2368**	**1541 (68%)**			**1706 (75%)**			**1639 (72%)**			**1485 (66%)**		
lua	4	1 (25%)	0	0	1 (25%)	0	0	1 (25%)	0	0	1 (25%)	0	0
php	16	2 (13%)	0	0	3 (19%)	1	0	3 (19%)	1	0	3 (19%)	0	0
libsndfile	18	7 (39%)	0	0	7 (39%)	0	0	7 (39%)	0	0	7 (39%)	0	0
libpng	7	3 (43%)	0	0	2 (29%)	0	0	3 (43%)	0	0	3 (43%)	1	0
libtiff	14	8 (93%)	2	1.00	6 (64%)	0	0	7 (71%)	1	0	7 (71%)	0	1.41
libxml2	17	4 (41%)	0	0	4 (35%)	1	0	4 (35%)	0	0	4 (35%)	0	0
sqlite3	20	4 (20%)	0	0	4 (20%)	0	0	4 (20%)	1	0	4 (20%)	0	0
openssl	20	3 (20%)	0	0	1 (10%)	0	0	3 (15%)	0	0	1 (10%)	0	0
TOTAL	**116**	**41 (35%)**			**34 (29%)**			**37 (32%)**			**36 (31%)**		

Table 15. Impact of making MCC and WM coverage objectives explicit in the code over the total number of unique bugs discovered by QSYM over five runs. Variance between the five runs is reported through the measured interquartile range (IQR) and standard deviation (SD).

Program	Bugs	QSYM			QSYM with MCC			QSYM with WM			QSYM with MCC + WM		
		Found bugs	IQR	SD	Found bugs	IQR	SD	Found bugs	IQR	SD	Found bugs	IQR	SD
base64	48	18 (38%)	1	0	19 (40%)	2	1	20 (42%)	1	1.41	17 (35%)	1	1
uniq	29	14 (48%)	1	2	10 (34%)	0	0	11 (38%)	3	1.73	12 (41%)	2	2
md5sum	57	24 (42%)	2	1.41	25 (44%)	1	0	25 (44%)	0	0	25 (44%)	0	0
who	2234	38 (2%)	1	4.36	40 (2%)	4	2.82	52 (2%)	1	4.9	50 (2%)	6	4.69
TOTAL	**2368**	**94 (4%)**			**94 (4%)**			**108 (5%)**			**104 (4%)**		
lua	4	1 (0%)	0	0	1 (25)	0	0	1 (0%)	0	0	1 (0%)	0	0
php	16	2 (19)	0	0	3 (19%)	0	0	3 (19%)	0	0	3 (19%)	0	0
libsndfile	18	7 (39%)	0	0	6 (33%)	0	0	6 (33%)	1	0	7 (39%)	1	0
libpng	7	3 (43%)	0	0	2 (29%)	0	0	2 (29%)	0	0	2 (29%)	0	0
libtiff	14	5 (36%)	1	0	2 (29%)	0	0	4 (29%)	0	0	4 (29%)	0	0
libxml2	17	2 (12%)	0	0	2 (12%)	0	0	1 (6%)	0	0	1 (6%)	0	0
sqlite3	20	2 (10%)	1	0	1 (5%)	1	0	2 (10%)	1	0	1 (5%)	0	0
openssl	20	2 (10%)	1	0	3 (15%)	0	0	3 (15%)	1	0	2 (10%)	0	0
TOTAL	**116**	**24 (21%)**			**23 (20%)**			**21 (18%)**			**20 (17%)**		