Improving Fuzzing through Controlled Compilation

Laurent Simon
Samsung Research America

Akash Verma
Samsung Research America

***Abstract*—We observe that operations performed by standard compilers harm fuzzing because the optimizations and the Intermediate Representation (IR) lead to transformations that improve execution speed at the expense of fuzzing. To remedy this problem, we propose 'controlled compilation', a set of techniques to automatically re-factor a program's source code and cherry pick beneficial compiler optimizations to improve fuzzing.**

We design, implement and evaluate controlled compilation by building a new toolchain with Clang/LLVM. We perform an evaluation on 10 open source projects and compare the results of AFL to state-of-the-art grey-box fuzzers and concolic fuzzers. We show that when programs are compiled with this new toolchain, AFL covers 30% new code on average and finds 21 additional bugs in real world programs.

Our study reveals that controlled compilation often covers more code and finds more bugs than state-of-the-art fuzzing techniques, without the need to write a fuzzer from scratch or resort to advanced techniques. This suggests that advances in fuzzing are more limited than previously believed.

We identify two main reasons to explain why. First, it has proven difficult for researchers to appropriately configure existing fuzzers such as AFL. To address this problem, we provide guidelines and new LLVM passes to help automate AFL's configuration. This will enable researchers to perform a fairer comparison with AFL.

Second, we find that current coverage-based evaluation measures (e.g. the total number of visited lines, edges or BBs) are inadequate because they lose valuable information such as *which* parts of a program a fuzzer actually visits and how *consistently* it does so. Coverage is considered a useful metric to evaluate a fuzzer's performance and devise a fuzzing strategy. However, the lack of a standard methodology for evaluating coverage remains a problem. To address this, we propose a rigorous evaluation methodology based on 'qualitative coverage'. Qualitative coverage uniquely identifies each program line in order to help understand which lines are commonly visited by different fuzzers vs. which lines are visited only by a particular fuzzer. Throughout our study, we show the benefits of this new evaluation methodology. For example we provide valuable insights into the consistency of fuzzers, i.e. their ability to cover the same code or find the same bug across multiple independent runs.

Overall, our evaluation methodology based on qualitative coverage helps to understand if a fuzzer performs better, worse, or is complementary to another fuzzer. This helps security practitioners adjust their fuzzing strategies.

I. Introduction

A fuzzer is made up of several components: a compilation toolchain for coverage instrumentation, an initial input generation engine, a seed scheduler engine, and a seed mutation engine [1], [2]. There has been considerable research on how to effectively generate initial input [3], [4], mutate seeds and adapt the seed scheduling strategy [5], [6], [7], [8], [9]. However, there is little to no research studying the compilation toolchain to improve fuzzing.

Compilers typically optimize for speed and size, rather than for fuzzing. Compiler optimizations have been shown to impede software security by removing important security checks in code [10], [11], or by violating developers' assumptions in cryptography implementations [12]. In this work, we study how a standard compiler toolchain also undermines fuzzing.

We observe that compiler optimizations and IR semantics hinder fuzzing (Section II). For example, we find that optimizations tend to merge basic blocks to improve execution speed, but this is harmful as it requires a fuzzer to satisfy multiple conditions in a single input mutation. Disabling optimization altogether seems like an easy solution but it slows down program execution. Undoing compiler optimizations (`LAF-INTEL` approach [13]) is a difficult task because compilers have large and complex codebases (e.g. LLVM has over 50 transformation optimizations [14]). So we propose 'controlled compilation', a set of techniques to re-factor source code and cherry pick beneficial compiler optimizations to make compilation fuzzing-friendly. We take the view, like previous research [12], that it is more reliable to inform the toolchain about what it should do, rather than trying to undo its optimizations.

We build controlled compilation into a new toolchain using Clang/LLVM. Using this new toolchain, AFL visits new lines on real world programs and finds 21 additional bugs compared to vanilla AFL (which finds 6 bugs). In comparison, recently published tools such as Angora and QSYM find 13 and 6 additional bugs respectively. This suggests that AFL is even more efficient than previously believed, given it is appropriately configured. Progress made in fuzzing techniques seems to be more limited than previously thought.

We believe this stems from two main reasons: First, it has been difficult for researchers to run AFL in its best configuration (e.g. with a good dictionary). So we provide guidelines and LLVM passes to automate this process (Section VI). This should enable a fairer comparison with AFL for future research.

The second reason is that current coverage metrics vary greatly across research papers and are often inadequate (Section IV). Although a fuzzer's ultimate goal is to find bugs, coverage is commonly seen as a useful proxy to build and evaluate a fuzzer. Widely-used fuzzers such as AFL or HonggFuzz [15] use coverage feedback to make progress and are based on the intuition that "no fuzzer can find bugs in code that is not covered" [8]. So coverage may be used to measure a fuzzer's performance and devise efficient fuzzing strategies. Coverage measures used in the fuzzing literature include the total number of lines, edges or basic blocks visited (Section IV). But these lose valuable information such as *which* parts of a program are visited. So we propose a new methodology based on 'qualitative coverage': in contrast to quantitative measures of coverage (e.g. raw number of lines), qualitative coverage uniquely identifies each program line. This allows us to more faithfully compare fuzzers to one another, for example we can identify lines commonly visited by different fuzzers vs. lines visited only by a particular fuzzer (Section V). Controlled compilation is a fundamental building block towards using qualitative coverage (Section IV): it allows us to generate a dedicated reference build optimized to capture each condition solved by a fuzzer's input. It also has the advantage of decoupling the binary used for fuzzing from the binary used for extracting coverage information.

Throughout our evaluation, we show the benefits of this new methodology. We study the consistency of fuzzers, i.e. their ability to visit the same lines and find the same bugs across multiple independent runs. We find this to be insightful to adjust the fuzzing strategy, for example for targeted fuzzing where the goal is to reach a particular set of lines (regression tests). We also highlight erroneous conclusions that may be drawn using quantitative coverage metrics (e.g. see `prettyprint` program in Section V).

We contribute the following:

- We observe that a standard compiler toolchain hinders fuzzing. We propose 'controlled compilation' to remedy these problems. By putting the source code and IR of the program in a state that is more amenable to analysis, controlled compilation improves a fuzzer's ability to cover new code and find more bugs.
- We propose a more rigorous evaluation methodology based on qualitative coverage measures extracted from a reference build. The reference build captures path conditions that are solved by a given tool in a more faithful way. The qualitative measure allows researchers to better compare their results to one another, including a fuzzer's consistency across runs.
- We design and implement controlled compilation with Clang/LLVM compiler toolchain and evaluate the impact of this modified toolchain with AFL as the fuzzer.
- We make our code available[1].

II. Compiler transformations harm fuzzing

White-box dynamic analysis techniques, such as symbolic execution, can solve complex path conditions but suffer from state explosion when they try to visit all possible paths. On the other side of the spectrum, black-box fuzzing requires no knowledge of the program under test. It scales well, but ultimately suffers from this lack of knowledge to be effective. Grey-box fuzzing lies in between white-box and black-box analysis: it trades accuracy for scalability, by randomly mutating input whilst keeping track of the bare minimum state necessary. The random mutations can be repeated so many times that it makes up for the lack of accuracy.

In the rest of this paper, we will use the AFL fuzzer [16] as an example. AFL is a grey-box fuzzer widely used in academia and industry. AFL is a coverage-guided fuzzer, meaning that it deems inputs 'interesting' if they trigger new code paths in a program. Coverage is determined through lightweight program instrumentation that records which basic blocks (BBs) are executed – a basic block, loosely speaking, is a series of instructions without branches. When a testcase triggers a new path, AFL saves it and then uses genetic algorithms to mutate it. This allows AFL to automatically discover inputs that trigger new internal states in the targeted program. Note that for AFL, a path is determined not just by the basic blocks visited, but also by their order. AFL keeps track of each basic block transition, which it calls an 'edge'. Each edge is given a unique 16-bit integer $\texttt{edge}_{\texttt{id}}$ wich uniquely identifies a transition $\texttt{BB}_{\texttt{src}} \rightarrow \texttt{BB}_{\texttt{dst}}$ (a.k.a. branch or edge).

Since coverage-guided fuzzers like AFL rely on coverage information and have a high dependency on the control flow of programs, it is important to understand how compiler transformations affect these two aspects of a program. A compiler, amongst other things, tries to minimize the number of operations and also tries to replace existing operations with simpler ones. After such transformations are performed, the resulting binary may have a different structure from the original source code. In the rest of this paper, we focus on Clang/LLVM as an example. In Clang/LLVM, transformations may be performed in the Intermediate Representation (a.k.a. LLVM passes) or during translation from IR to machine code (a.k.a. in the backend passes). There are around 50 LLVM transformation passes [14], depending on optimization levels and options passed to the compiler.

In the following sections, we show several examples of transformations performed by LLVM that harm fuzzing. We explain, at a high-level, how these transformations affect the layout of basic blocks that coverage-guided fuzzers rely on, and ultimately hinder fuzzing. This is by no means an exhaustive list, but aims at supporting the claim that compiler transformations do harm fuzzing.

A. Condition Merging

The main transformation that affects the layout of basic blocks is the merging of conditions. It changes the Control Flow Graph (CFG) and harms fuzzing. Condition merging takes place in several transformations, as we explain next.

Block-level Optimizations:

A major reason for merging condition at the IR level is to simplify other LLVM passes, by enabling them to work at the level of a single basic block (`BB`). The LLVM pass responsible for merging `BB`s is called `SimplifyCFG` [14]. For example, consider the code in Listing 1 on the following page. The resulting IR before optimization is depicted in Listing 2 on the next page: the IR contains four `BB`s. In basic block `BB_1`,

[1] https://github.com/Samsung/afl_cc

Listing 1: Original source code

```
int result = 0;
if (x == MVx) {
    int w = x + y;
    if (y == MVy) {
        int z = x + y;
        result = z + w;
    }
}
return result;
```

Listing 2: LLVM IR before optimizations

```
1 // BB_0
2 result = 0
3 if x == MVx: goto BB_1 else BB_end
4
5 // BB_1
6 w = x + y
7 if y == MVy: goto BB_2 else BB_end
8
9 // BB_2
10 z = x + y          /* z has the same value as w */
11 result = z + w
12 goto BB_end
13
14 // BB_end:
15 ret result
```

Listing 3: LLVM IR after `simplifyCFG`

```
1 // BB_0
2 result = 0         /* conditions merged */
3 cxy = (x == MVx) && (y == MVy)
4 if cxy: goto BB_12 else BB_end
5
6 // BB_12
7 w = x + y
8 z = x + y
9 result = z + w
10 goto BB_end
11
12 // BB_end:
13 ret result
```

Listing 4: Switch-statement

```
switch(a) {
  case 1:
    return 123;
  case 2:
    return 456;
  case 3:
    return 789;
  default:
    return 0;
}
```

variable `w = x + y` is defined (line 6); and the same value is copied into variable `z` in basic block `BB_2` (line 10). It is obvious that both variables have the same value `x + y`, so it makes sense for the Common Subexpression Elimination (CSE) to replace these two computations with a single one. During compilation, the `SimplifyCFG` pass merges `BB_1` and `BB_2` into `BB_12` (Listing 3). A `CSE` pass that only works at the level of a basic block can then easily spot that `z` and `w` store the same value, and replace the two computations with a single one.

Unfortunately, merging `BBs` has unintended consequences with regards to fuzzing. Consider an input where $\mathtt{x} \neq \mathtt{MV}_\mathtt{x}$ and $\mathtt{y} \neq \mathtt{MV}_\mathtt{y}$: the input visits only `BB_0` and `BB_end` (Listing 2). If a different input is such that $\mathtt{x} = \mathtt{MV}_\mathtt{x}$ and $\mathtt{y} \neq \mathtt{MV}_\mathtt{y}$, then the input visits `BB_1` in addition to `BB_0` and `BB_end`: therefore AFL figures that the input improves coverage and saves the seeds. Similarly, AFL will mutate this seed to find new inputs that visit all `BBs`.

However, when `SimplifyCFG` runs (typically when optimization is enabled), this works differently, as shown in Listing 3. When $\mathtt{x} = \mathtt{MV}_\mathtt{x}$ and $\mathtt{y} \neq \mathtt{MV}_\mathtt{y}$, no additional `BBs` are visited because the conditions `if (x == MV`$_\mathtt{x}$`)` and `if (y == MV`$_\mathtt{y}$`)` are now merged into a single `if (x == MV`$_\mathtt{x}$ `&& y == MV`$_\mathtt{y}$`)` (`BB_0`, line 3 in Listing 3). This means that AFL must find values for both `x` and `y` at the same time in order to satisfy both conditions. This is much harder: assuming a random mutation of the input, the probability of success for two 16-bit integers would be $2^{-16} \times 2^{-16}$ rather than $2^{-16} + 2^{-16}$.

Branch-less Optimizations:

Another reason to merge conditions or remove branches in programs is speculative execution. Each time the CPU encounters a conditional branch, it makes an educated guess about which basic blocks to prefetch next. If the guess turns out to be wrong, the pipeline is halted and this incurs a performance hit. Therefore, by reducing the number of conditional branches in a program, the number of guesses the CPU has to make decreases, and the speed improves.

There are several optimization techniques used by LLVM to remove conditional branches, including bitmaps, linear maps, and table lookups. An example for a `switch`-statement is provided in Listing 4; and its resulting (optimized) IR in Listing 5 on the next page. In the optimized IR code (Listing 5), `a`-dependent branches are removed. Instead, the return value is stored in an array `switch_table` (line 2) and retrieved through a branch-less lookup (line 8). This harms fuzzing for the same reasons we explained before: a coverage-guided fuzzer will miss interesting inputs that return different values, since different inputs will not alter the control flow.

Single Instruction Multiple Data (`SIMD`) instructions (e.g. AVX on x86) allow performing a comparison over multiple bytes in a single instruction. They offer an even more aggressive solution to reduce branches and improve performance, but at the expense of fuzzing.

Built-in functions:

Another transformation used by compilers is to replace `libc` functions (e.g. `strcmp()`, `strncmp()`, etc.) with built-in functions (e.g. `memcmp()`). This allows the compiler

Listing 5: LLVM IR generated with Clang `-O3`

```
// BB_start
  switch_table[3] = {123, 456, 789}
  a = ...
  a -= 1
  if a < 3: goto BB_lookup else: goto BB_def

// BB_lookup
  ret = switch_table[a]
  return ret

// BB_def
    return 0;
}
```

Listing 6: Three integer comparisons

```
if (x==0x01 && y==0x02 && z==0x03)
    <some code>
```

to inline the code and remove branches, which helps to re-use the ideas presented in previous sections. For example, a call to `strcmp()` may be replaced by a call to `memcmp()`, which may then be converted into a `SIMD` instruction. However, this further decreases the probability of finding all the right magic values. For example for Intel AVX vectorization instructions which support up to 256-bit registers, the probability would go down to 2^{-512}.

B. *LLVM vs. binary coverage*

Compiler internals do not just harm fuzzing, they may also harm other kinds of runtime analysis such as concolic execution engines. Compiler transformations may introduce differences between the `BB` layout in the IR vs. the `BB` layout in the final binary. This creates problems for concolic engines that expect the LLVM `BB` and the final binary's `BB` layout to be identical. Take as example the code in Listing 6. When compiled with optimizations by AFL's toolchain, conditions are merged as shown in Listing 7: there are three `BB`s. Now consider the resulting binary generated (Listing 8): there are five `BB`s. There is inconsistency between the binary and the LLVM IR regarding basic block layout. What happens is that the compiler backend (which converts IR instructions into machine code) has transformation passes of its own. Using certain heuristics and architecture knowledge, it determines that inserting additional branches for early exits is advantageous for this platform. But this creates conflicts. AFL's view of coverage is implemented at compilation time on the LLVM IR representation, so basic blocks are merged as depicted in Listing 7. A concolic engine (e.g. QSYM [17]) that uses the binary for symbolic input replay sees a different view of coverage as depicted in Listing 8. Therefore, when such a concolic engine generates new input that satisfies additional conditions in the binary, AFL may discard it because it expects all conditions to be satisfied at once in the IR. We verified this to be an issue for `QSYM`.

Listing 7: LLVM IR when compiled with Clang `-O3` of Listing 6

```
// BB_0
    cx = (x == 0x01);
    cy = (y == 0x02);
    cz = (z == 0x03);
    cxy = cx && cy
    cxyz = cxy && cz
    if cxyz: goto BB_code else goto BB_end

// BB_code
    <some code>
    goto BB_end

// BB_end
...
```

Listing 8: Resulting binary when compiling the code of Listing 6 with gcc and Clang `-O3` for `x86-64`

```
// BB_0
    cmpl $0x01, x
    jne BB_ret

// BB_1
    cmpl $0x2, y
    jne BB_ret

// BB_2
    cmpl $0x3, z
    jne BB_ret

// BB_code
    <come code>

// BB_ret
    ...
```

III. The Need for Controlled Compilation

We initially thought that disabling all compiler transformations was the best way to eliminate all these problems. However, after conducting some preliminary experiments, we found this was not necessarily the case. In the following sections, we take three examples as case studies. The takeaway message of this section is that neither disabling nor enabling all compiler optimizing transformations is satisfactory. This motivates 'controlled compilation', a way to inform the compiler about which transformations to perform.

A. *Predicated Instructions*

Clang/LLVM makes use of predicated instructions both at the IR level and in the backend, and this may happen even when optimizations are disabled. Predication helps speed up execution by converting control dependence into data dependence to eliminate branches. For example, consider the code using the conditional operator `res = condition ? cond_true : cond_false`, which is functionally equivalent to the `if`-statement `if (condition) res = cond_true; else res = cond_false`. The

Listing 9: Four-byte comparison of integer

```
int16_t a = ...;
if (a == 0x1234)
    abort();
```

Listing 10: Transformation of four-byte comparisons (Listing 9) into a series of four single-byte comparisons. We assume Big-Endianess

```
int16_t a = ...;
uint8_t * ptr = &a;
if (ptr[0] == 0x12)
    if (ptr[1] == 0x34)
        abort();
```

former, using the conditional operator is compiled by Clang/LLVM into the branch-less, predicated instruction called `select()` as `res = select(condition, cond_true, cond_false)`, regardless of optimizations used. The IR instruction `select()` is typically lowered into a branch-less, constant-time `CMOV` instructions by the LLVM backend. Unfortunately, the `select()` instruction removes branches depending on variable `condition` in the `CFG`. For the reason we described in previous sections, this harms fuzzing. Note that the backend also has a dedicated pass called `IfConversion` [18] responsible for transforming conditional branches into predicated instructions.

B. Byte-Splitting Passes (`BSP`)

A 'Byte-Splitting-Pass' (`BSP`) is an LLVM pass that transforms single multi-byte comparisons into multiple byte-by-byte comparisons. Consider the code of Listing 9. Using `BSP`, the goal is to transform the code into the code of Listing 10. The idea is that instead of trying to guess, say, a 16-bit value at once (probability 2^{-16}), `BSP` breaks each comparison into two single-byte comparisons. This way the fuzzer can incrementally guess each value (probability of 2×2^{-8}). This idea may be applied to larger integers too, and has been implemented by the authors of `LAF-INTEL` [13].

Consider the code in Listing 11, which performs a single one-byte comparison with a magic byte `MV`. Before the code is processed by `BSP`, the LLVM IR is as shown in Listing 12. Integer promotion is performed by the compiler (as part of the C standard), so integers smaller than four bytes are converted to four-byte integers prior to comparison. Then, when `BSP` runs, it (re-)splits each four-byte comparisons into a series of single-byte comparisons (Listing 13 on the next page). We end up with four byte-comparisons instead of one as in the original code, which may decrease program execution speed and increase the probability of edge collision. Recall that for AFL, an edge is a unique integer $\texttt{edge}_{\texttt{id}}$ identifying a transition from a $\texttt{BB}_{\texttt{src}} \rightarrow \texttt{BB}_{\texttt{dst}}$, and can only take 64k distinct values. A collision occurs when two different edges end up having the same $\texttt{edge}_{\texttt{id}}$. If an input triggers a new edge but its $\texttt{edge}_{\texttt{id}}$ is the same as another already-seen edge, AFL will incorrectly discard the input. Note that

Listing 11: Byte comparison

```
int8_t a = ...;
if (a==MV)
    abort();
```

Listing 12: Byte comparison without optimization and before BSP

```
int8_t a = ...;

int32_t a_32 = a;    // integer promotion
int32_t MV_32 = MV;  // integer promotion

if (a_32==MV_32)
    abort();
```

naïvely increasing the range an $\texttt{edge}_{\texttt{id}}$ can take to more than 64k values is not a viable solution, because it has the side effect of increasing cache misses and slowing down program execution [1].

C. Magic Value Extraction

AFL has an option that lets developers pass a set of magic values to use for fuzzing. This simple feature helps AFL satisfy comparisons that use hard coded values. Generating this set of magic values typically requires expert knowledge and/or reading through a format's standards, which is time consuming. An alternative to this human-generated dictionary is to create it automatically. One approach used today is extracting all hexadecimal-encoded values from a binary and using these as magic values. The number of magic values extracted this way may be overwhelming though. For example when extracting magic values for the `objdump` binary, over $300k$ values are returned. This ends up hurting fuzzing because the dictionary is large and contains many useless values. Instead, in this pass, we look at the possibility of generating the dictionary automatically at compilation time.

Consider the code in Listing 14 on the following page. After integer promotion is performed, the code looks like in Listing 15 on the next page. If we try to extract magic values from the program IR by scanning the constant operands of comparison instructions, we end up with two four-byte magic values: `MV0_32` and `MV1_32` (line 6 and 7 respectively in Listing 15 on the following page). What happens when AFL attempts to use the dictionary values is depicted in Fig. 1 on the next page. AFL successfully passes the first comparison with `MV1_32` (Step 1). Unfortunately, when trying the second magic value `MV0_32` (Step 2), it overwrites the bytes of `MV1` in the input and ends up failing on the first comparison. As a result, AFL fails to find a path to the `abort()` using the dictionary. On the contrary, if we control integer promotion, the magic values we extract are the original one-byte values `MV1` and `MV0`: these allow AFL to effortlessly satisfy the two comparisons. Note that this behavior may be controlled with LLVM's `-argpromotion` option.

Listing 13: Byte comparison without optimization and after BSP

```c
int8_t a = ...;

int32_t a_32 = a;    // integer promotion
int32_t MV_32 = MV; // integer promotion

// BSP splits the 4-byte comparison
// into 4 single-byte comparison
int8_t a0_8 = *((int8_t*) &a_32);
int8_t a1_8 = *((int8_t*) &a_32);
int8_t a2_8 = *((int8_t*) &a_32);
int8_t a3_8 = *((int8_t*) &a_32);

int8_t MV0_8 = *((int8_t*) &MV_32);
int8_t MV1_8 = *((int8_t*) &MV_32);
int8_t MV2_8 = *((int8_t*) &MV_32);
int8_t MV3_8 = *((int8_t*) &MV_32);

// We end up with 4 times more comparisons
// than necessary
if (a0_8==MV0_8)
    if (a1_8==MV1_8)
        if (a2_8==MV2_8)
            if (a3_8==MV3_8)
                abort();
```

Listing 14: Two byte-comparisons

```c
uint8_t * input_buffer = ...
uint8_t a0 = input_buffer[0];
uint8_t a1 = input_buffer[1];
if (a1==MV1)
    if (a0==MV0)
        abort();
```

Other useful ways to control compilation for dictionary generation include `constant folding`, a compiler technique that evaluates constant expressions at compilation time rather that computing them at runtime. For example, it transforms $2+(3\times 5)$ into 17, which makes it easier for us to extract the magic value 17.

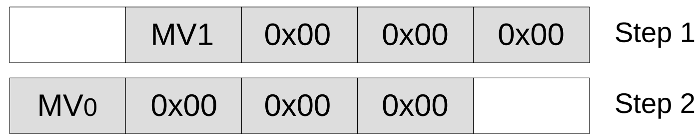
IV. Controlled Compilation Helps Fuzzing Evaluation

The previous sections highlighted the potential benefits of controlled compilation to improve a fuzzer's ability to pass conditions. In this section, we show that controlled compilation is also beneficial for fuzzer evaluation. There is consensus in the community that we need to improve the evaluation of fuzzing research [30], [31]. So we decided to survey previous research (totalling 38 papers) to learn how coverage evaluation is conducted in the fuzzing literature. Recall that coverage is commonly seen as a proxy to evaluate fuzzers because "no fuzzer can find bugs in code that is not covered" [8]. In previous research, about half of the papers use coverage to measure their tool (TABLE I). Among these, coverage evaluation varies greatly. In general, we find that fuzzers do not quantify which things they have to do worse in order to improve others. This is unlike in other fields, like intrusion detection or machine learning, where tradeoffs are explicitly acknowledged, e.g. via RoC curves, `F1` scores, recall, precision, etc. Next, we present some on the insights we gained while reviewing previous research's coverage evaluation. This will motivate the need for harmonizing coverage metrics and for a more rigorous evaluation strategy.

Listing 15: Two-byte comparison after integer promotion

```c
uint8_t * input_buffer = ...
uint8_t a0 = input_buffer[0];
uint8_t a1 = input_buffer[1];
uint32_t a0_32 = a0;  // each 8-bit variable
uint32_t a1_32 = a1;  // is converted to
uint32_t MV0_32 = MV0;// a 32-bit integer
uint32_t MV1_32 = MV1;
if (a1_32==MV1_32)    // comparisons of
    if (a0_32==MV0_32)// 32-bit integers
        abort();
```

Fig. 1: AFL's use of the dictionary for code of Listing 14. We assume Little Endianness.

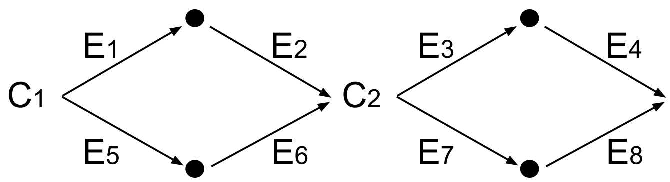


fuzzer	coverage measure	multiple runs	duration >=24H	parallel fuzzing	resource consumption
QSYM [17]	L, BE	Y	N	Y	N
Steelix [19]	P, L, F, SE	N	Y	N	N
Angora [5]	L, SE	Y	N	N	N
NEUZZ [20]	BE	N	Y	N	N
Redqueen [8]	BBB	N	N	N	N
MutaGen [21]	I	Y	Y	N	N
Driller [22]	BBB, P	N	Y	Y	N
AFLGo [23]	SBB	Y	1 target	N	N
Skyfire [3]	L, F	N	Y	N	N
kAFL [24]	P	Y	N	N	N
IMF [25]	API	N	Y	N	N
FairFuzz [26]	SE	Y	Y	N	N
CollAFL [1]	P	Y	Y	N	N
MOpt [27]	P	Y	Y	Y	N
SmartSeed [4]	P	N	Y	N	N
AFLSmart [28]	P	Y	Y	N	N
V-Fuzz [29]	BBB	N	Y	N	N

TABLE I: Fuzzers's setup and coverage evaluation. N = No, Y = Yes. BE = Binary edges (taken from AFL's bitmap in QEMU mode), SE = source edges (taken from AFL's bitmap in LLVM mode or with `afl-cov` or/`gcov`), P = (AFL's reported) Paths, BBB = Binary Basic Blocks, SBB = Source Basic Blocks, L = Lines, F = Functions (taken from source code), API = Application Programming Interface. Resource consumption excludes execution speed.

Unique paths: Almost half of research papers that report coverage use the number of 'unique paths' as a measure of

Fig. 2: Example of a CFG. C_i are conditions; E_i are edges.



coverage ('P' in the first colum of TABLE I on the previous page). The unique path is a metric used by AFL, and it represents the number of inputs (a.k.a. seeds) that AFL saves because they are considered interesting. The number of paths has nothing to do with coverage per-se and is not appropriate for reporting coverage. Take as example the control flow graph depicted in Fig. 2. Assume a fuzzer F_1 that finds inputs that satisfy all four conditions (condition C_1 branches to edge E_1 and E_2; condition C_2 branches to edge E_3 and E_7). Assume F_1 satisfies the four conditions with two inputs: input I_1 follows path $E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4$, and input I_2 follows path $E_5 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8$. Now imagine a fuzzer F_2 that satisfies only three conditions but with three inputs: input I_1 follows path $E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4$; input I_2 follows path $E_1 \rightarrow E_2 \rightarrow E_7 \rightarrow E_8$ and input I_3 follows path $E_5 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8$. Using the number of unique paths as a measure of coverage, we would conclude the second fuzzer F_2 satisfies more conditions. This is incorrect though, since it fails to find an input that takes edge E_7 of condition C_2. This demonstrates how the number of unique paths is inadequate. Using lines, basic blocks or edges is better suited to report coverage because these directly relate to the conditions solved by a fuzzer.

Binary vs. source code coverage metrics:

Over half of the papers report the number of basic blocks (Binary BB 'BBB' or Source BB 'SBB') or edges (Source edges 'SE' or Binary edges 'BE') visited by their fuzzer (first column of TABLE I on the previous page). This has limitations too, because these measures depend on the toolchain used to compile a program. Since compilers have different transformations and optimizations, etc., edges and basic blocks differ for binaries generated by different compilers (AFL, for example, supports both gcc and Clang to compile a program). This problem is aggravated when comparing a binary-instrumented fuzzer vs. a source code-instrumented one. As we showcased in Section II-B on page 4, the layout of basic blocks differs between the IR (used by source code instrumentation) and the generated binary (used by binary fuzzers). This creates inconsistency with regards to the ground truth coverage to compare fuzzers faithfully. In the literature, around one third of papers report binary-based numbers ('BE' or 'BBB') while the remaining two third report source-based numbers ('SE' or 'SBB'). We believe harmonizing coverage metrics would help researchers by having a common 'reference' build for the sole purpose of extracting coverage information.

Quantitative vs. Qualitative Coverage:

Listing 16: Single if-statement

```
int a1 = ...;
int a2 = ...;
int a3 = ...;

if (a1==1 && a2==2 && a3==3) <some code>
```

Listing 17: Multiple if-statements

```
int a1 = ...;
int a2 = ...;
int a3 = ...;

if (a1==1)
    if (a2==2)
        if(a3==3)
            <some code>
```

A major problem in evaluating and comparing fuzzers with regards to coverage is that the research community has only considered quantitative metrics rather than qualitative ones. With the exception of AFLGo [23], previous research only reports the total number of lines, basic blocks, and/or edges visited by their tool. But this raw number is too coarse: for example, it cannot capture if there is overlap between the lines visited by different fuzzers. It is possible that different fuzzers visit a similar number of lines but visit different parts of the program (we will see examples of this in the evaluation section). One can be mislead into concluding that two fuzzers perform similarly if the coverage metric cannot capture this difference in behavior. Therefore, we argue the community should switch to using a qualitative coverage metric, i.e. a measure where each line is uniquely identified. But for this, we need a common definition of coverage.

Definition of Coverage:

A major obstacle towards using a qualitative coverage metric is that it requires a common definition of what is a line, a basic block, or an edge. For example, for basic blocks, shall we use the IR-level view or the binary-level view? Or shall it be something else entirely?

Consider the code in Listing 16. Assume we want to compare two fuzzers F_1 and F_2. Assume F_1 does not find inputs that satisfy the conditions `a1==1` and `a2==2`, but F_2 does: obviously F_2 is better than F_1. However, if we plot their line coverage, both fuzzers appear equivalent as they visit the same lines. To remedy this problem, we propose re-factoring the code when compiling a reference build for extracting coverage information.

Intuitively, we should re-factor the original code into the code in Listing 17: each condition appears on a different line so it will be reflected in the line coverage. The guiding principle is that every line should represent one statement. For example `Var1 BinOp Var2` should always appear on a single line. This may seem obvious but it is not always the case in source code. We can perform code re-factoring in the compiler in order to generate a reference build for extracting coverage information. Then, to compare fuzzers, we simply run the reference build with each fuzzer's generated inputs. We end up with a list of line/edge/BB numbers corresponding to visited code statements. Note that the code re-factoring may be performed at various levels of granularity: for example a statement like `tab[a++]` may be rewritten on two lines; `strcmp()` may be broken into byte-by-byte, or even bit-by-bit comparisons. In our current prototype, we keep these as in the source code; we show through evaluation this is useful as is.

When re-factoring the code, we must be wary of compiler transformations, as these may undo some of the code-refactoring changes we apply. So we re-use ideas from previous sections to tune which optimizations to run after the code has been re-factored. This ensures that the characteristics added to the source code are reflected in the IR.

Because a qualitative coverage metric captures information about exactly which parts of a program are visited, it is particularly useful for regression testing/targeted fuzzing where the goal is to reach a particular set of lines. In this scenario, knowing which fuzzer is better at consistently reaching which lines helps to select the right fuzzer.

Summary: Qualitative coverage labels each statement/-line/BB with a unique ID. It forms the fundamental building block of our evaluation methodology. Throughout our evaluation (Section V), we use the definition of qualitative coverage as defined above. Qualitative coverage lends itself to the study of fuzzer consistency, i.e. how consistent a fuzzer is at visiting the same lines across multiple independent runs, as we shall see in the next sections.

V. Evaluation

A. Methodology based on qualitative coverage

As we described in the previous section, we use qualitative coverage as the basis of our coverage evaluation methodology. Previous research reports a quantitative measure of coverage (e.g. number of lines); and some papers additionally report its standard deviation. Imagine a fuzzer that visits different parts of a program on each run, yet always visits about the same number of lines. Reporting the number of lines along with its standard deviation would lead to the conclusion that the fuzzer is consistent, which is incorrect.

With qualitative coverage, we no longer use the number of lines visited. Instead we uniquely identify which lines are visited and we compare these to another fuzzer. However, it is important to realize that we can no longer use a single standard deviation because we have a huge number of different lines, not just a single raw line number. Therefore we must devise a different strategy to compare fuzzers, which we describe now. There are two relevant questions a security practitioner wants to answer when benchmarking a fuzzer:

1. If a fuzzer runs N times, what is the maximum coverage (i.e. which lines will it visit)? This is a valuable question to ask because security practitioners run fuzzers many many times. They want to understand the commulative lines (or bugs) a fuzzer will visit. To report this, we use histogram-based cumulative coverage graphs (CCG). Because the graphs report cumulative coverage, there is no standard deviation depicted in CCGs (we shall see an example in the next sections).

2. Across these N runs, how consistent is the fuzzer? For this we compute a coverage consistency score (CCS) which represents a fuzzer's consistency, i.e. how often lines/bugs are encountered across all independent runs. Take the example of line consistency. For each line L_i, its consistency C_i ($0 \leq C_i \leq 100\%$) is the ratio of the number of runs that visit the line L_i over the total number of runs. For example, if we launch ten independent runs and four runs only visit line L_i, then line L_i has a consistency $C_i = 4/10 = 40\%$. We then use each line's consistency to generate the final CCS. We initially thought of using the mean of all line's consistency along with a standard deviation. However, because the consistency does not follow a normal distribution (this will become obvious with examples in the next sections), we instead decided to use the median along with the median absolute deviation (MAD; which represents the spread/deviation of the consistency values from the median). So the CCS is comprised of two numbers: the median of line consistencies and the MAD. A perfect CCS is $(100\%, 0)$, i.e. all visited lines are visited in all runs (100% of the time). In addition to the CCS, we also plot a 'consistency graph' which is a visual depiction of a fuzzer's consistency. The CCS is a compact representation of consistency, and the graph is more illustrative as we shall see in the next sections.

B. Implementation and Setup

We implement a new toolchain using Clang/LLVM: it supports compiling a program *1*) for fuzzing and *2*) for coverage extraction. Code-refactoring is implemented using a custom `clang-format` and custom Clang plugins.

Our LLVM pass for automated dictionary generation extracts constant integers used as operands to (non-)equality comparison instructions; constants and constant structure fields passed as argument to function calls; and constants (arrays, strings) used in memory comparison functions (e.g. `memcmp()`, `strcmp()`, etc.). We also implement what we call an 'optimized dictionary' where we not only extract magic values, but also the basic blocks where they are used. This way we can inform a fuzzer about which magic values are relevant for the current input/path under mutation. For the Byte-Splitting-Pass (BSP), we implement two variants: *1*) Full BSP (FBSP), which transforms all comparisons into a series of byte comparisons; and *2*) Light BSP (LBSP), which only transforms comparisons if the operand is not a magic value recorded by the dictionary extraction pass. Our intuition was that splitting a multi-byte comparison may not be necessary if a fuzzer has a magic value to pass this comparison anyway. Note that LAF-INTEL is different from FBSP and LBSP, as it splits bytes only for comparisons that use a hardcoded operand.

Binaries compiled with optimizations run fast but suffer from problems presented in Section II on page 2, such as BB merging, etc. Binaries compiled with controlled compilation

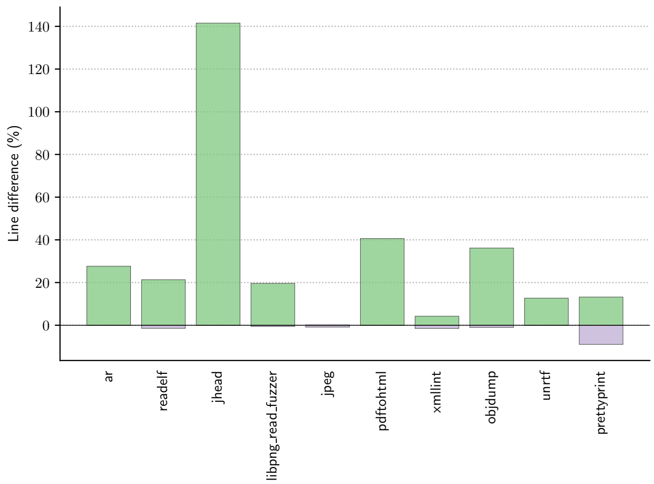


Fig. 3: Comparison of cumulative line coverage of C_FBSP_OD vs. baseline V_O3 (vanilla AFL).

are slower since some optimizations are disabled, but provide better coverage feedback for a fuzzer to make progress. Slower still are binaries compiled with Byte-Splitting Passes (BSP), since the passes further increase the size of the binary and the number of branches. But they arguably have the best feedback for a fuzzer to make progress.

Therefore, we see that each binary has pros and cons. So we decide to run them in parallel to take advantage of each of them. We call each of these binaries an 'AFL Companion' (AFLC). We run three companions in parallel and we ensure they share the inputs they find at runtime.

C. *Results on Real World programs*

We wanted to have as much diversity as possible, including programs that manipulate audio/multimedia, parse binary/text formats and that perform compression. So we settled for the following 10 real-world programs (see Appendix for arguments used): ar (compression); jhead, jpeg and png (image parsers); xml, html and unrtf (text parsers); objdump, readelf and pdf (binary-format parsers). We run 17 fuzzing configurations, each being a combination of three AFL companions running in parallel - a detailed description of each is given in TABLE III on page 15 in Appendix. We run each configuration 24 hours; and we launch 10 independent runs. We distribute the fuzzing workload across 8 machines running Ubuntu with 16GB of RAM each. It took around 8 months to complete the experiments. We answer the following research questions based on the results from our experimental setup:

RQ1: Does controlled compilation boost fuzzing?

Cumulative coverage across all runs:

Consider Fig. 3 which shows the cumulative line coverage graph (CCG) of C_FBSP_OD vs. baseline V_O3 (vanilla AFL). A histogram marked N/A means the fuzzer could not be tested because of limitations of the current implementation of the tool. Overall, controlled compilation helps AFL to visit more code than vanilla AFL (V_O3) for seven programs out of ten:

Listing 18: elf.c for objdump

```
static bfd_boolean                                          10556
  elf_parse_notes (bfd *abfd, char *buf,
                   size_t size, file_ptr offset)
    ...

  else if (in.namesz == sizeof "stapsdt"                    10632
   && strcmp (in.namedata, "stapsdt") == 0)                 10633
    {
      ...

}
```

Listing 19: char_ref.c in prettyprint

```
static bool consume_named_ref(
    struct GumboInternalParser* parser,
    Utf8Iterator* input, bool is_in_attribute,
    OneOrTwoCodepoints* output) {
    ...
switch ( *_acts++ ) {

  case 1218:
    ...

  case 1990:
    ...

  case 2071:
    ...
```

C_FBSP_OD visits 140% new lines for jhead (histogram above the horizontal zero line); 40% for pdftohtml; around 30% for ar and objdump; 20% for readelf and libpng and 10% for unrtf, whilst missing a negligible number of lines seen by the baseline V_O3 (histogram below the horizontal zero line). For prettyprint, 15% new lines are visited, but close to 10% of lines visited by the baseline are missed (histogram below zero). After investigation, we found that the code of prettyprint defines html magic values with a unique string containing a list of tags as "<html>|<\html>|<img>|<\img>...". This string is parsed at runtime to extract the tokens separated by the | character. Because of this, our automated dictionary routines cannot extract the magic html tokens (since the operands to the comparison instructions do not appear constant). Note that had we used a quantitative coverage measure such as the total number of lines visited, we would have concluded that C_FBSP_OD and V_O3 are equivalent on prettyprint, since both visit the same number of total lines. However, with qualitative coverage, we find these two fuzzers are complimentary.

All cumulative line coverage results with vanilla AFL are availble in Appendix in Fig. 12 on page 19. Note that all fuzzing configurations have a coverage curve that flattens out after a few hours. Since this is a well-known behavior of fuzzers, we provide only one example for objdump in Fig. 6 on page 15 (Appendix).

Some examples of lines visited with controlled compilation are depicted in Listing 18 on the previous page: the condition at line 10632 is satisfied only by fuzzers that use controlled compilation: this is because without controlled compilation, the two conditions at line 10632 and line 10633 are merged so a fuzzer cannot tell when the first condition is satisfied by an input. Both conditions are satisfied only by fuzzers that use both controlled compilation and an automatically generated dictionary (C_⋆AD and C_⋆OD). Another example of code is shown in Listing 19 on the preceding page, where the case statements are only satisfied for fuzzers that use FBSP and controlled compilation (C_FBSP_⋆).

Cumulative bugs found across all runs:

For the analysis of bugs, we implemented a crash analyzer to compare the stack traces generated by gdb for all reported crashes, and then performed a manual analysis of candidates to identify the final list of distinct bugs found. For four programs (djpeg, prettyprint, libpng_read_fuzzer, and xmllint), no bugs are found by any of the configurations. For the remaining six programs, 36 distinct bugs are found across all fuzzing configurations. Except for unrtf, across the 10 runs, all fuzzing configurations find a superset of the bugs found by vanilla AFL. TABLE VI on page 17 shows the classification of discovered bugs by type and by fuzzer. C_FBSP_OD finds 27 of the 36 total bugs discovered. This is 21 more bugs than those found by baseline V_O3, 14 more than QSYM, 8 more than Angora, 13 more than LAF-INTEL and 23 more than MOpt.

RQ2: How does controlled compilation perform vs. concolic execution?

To answer this question, we look at QSYM, the state-of-the-art concolic execution engine at the time of writing. Cumulative coverage for C_FBSP_OD vs. QSYM is presented in Fig. 7 on page 16 (the baseline used is QSYM). For several binaries, controlled compilation covers over 20% new lines and misses fery few lines visited by QSYM. For the other binaries, the benefits are more nuanced: for example for objdump, controlled compilation visits 9% more lines than QSYM but misses 2% of the lines QSYM visits. For xmllint, QSYM and C_FBSP_OD visit the same number of lines, but each misses 2% of the lines visited by the other. Since both fuzzers perform similarily on xmllint, let us look at consistency graphs as shown in Fig. 4 on the next page. The left-most histogram represents the number of lines visited 100% of the time: for C_FBSP_OD (Fig. 4b on the following page), 83% of lines are visited 100% of the time. For QSYM (Fig. 4a on the next page), only 55% are visited 100% of the time; and about 20% of the lines are visited 50% of the time. The Coverage Consistency Scores (CCS) for QSYM and C_FBSP_OD are $(100, 33)$ and $(100, 11)$ respectively, which confirms C_FBSP_OD is more consistent than QSYM for xmllint. Note that in most cases, fuzzers have the first value of the CCS as 100 because the majority of the lines are easy to reach (they are visited in 100% of the runs). The MAD tells us about the 'spread' of the line consistency values from the median: in our example C_FBSP_OD has a lower MAD so is more consistent. The result is statistically significant according to the Mann-Whitney U test (p-value < 0.01).

In terms of cumulative bugs, C_FBSP_OD and QSYM find 13 common bugs. C_FBSP_OD finds 14 bugs not found by QSYM, and QSYM finds no bugs that C_FBSP_OD cannot find. All the common bugs found by the two fuzzers and their corresponding consistencies are shown in Fig. 13 on page 20. Each histogram represents a unique (common) bug found by both fuzzers and its height represents the consistency with which the bug is found (100 is the highest possible value which means the bug is found 100% of the time across all independent runs). Of the 13 common bugs depicted in the plot, 8 are found equally consistently by both fuzzers (bugs $4-11$); 3 more consistently by C_FBSP_OD (bugs $1-3$) and 2 more consistently by QSYM (bugs $12-13$).

The fact that concolic execution performs worse than fuzzing was surprising to us as we expected concolic execution to be more effective than fuzzing in general.

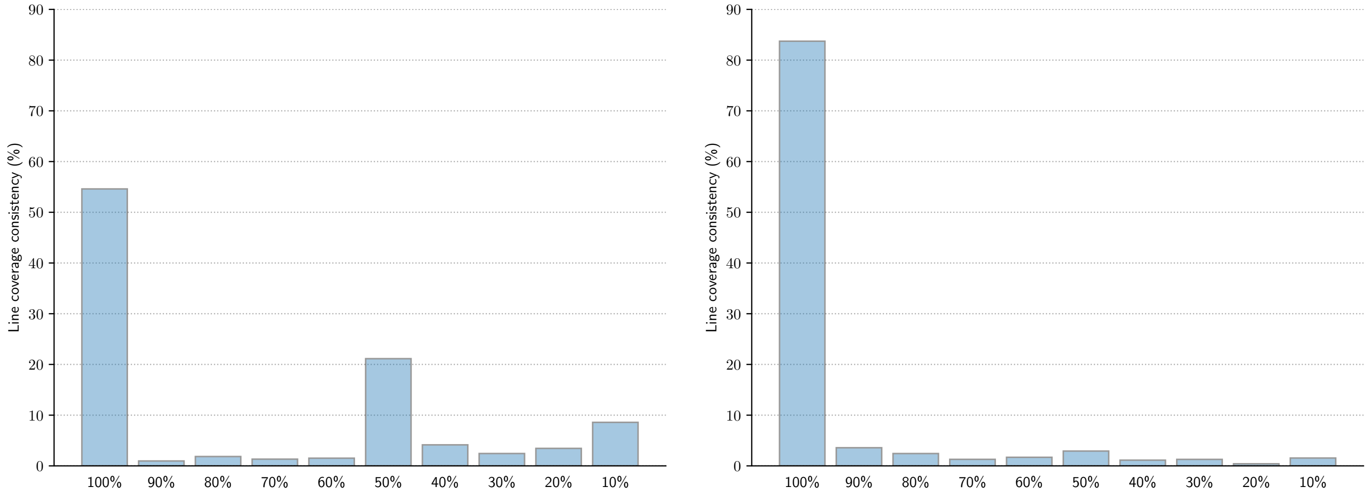
RQ3: How does an automatically-generated dictionary perform vs. a handcrafted one?

The configuration using the manually generated dictionary with vanilla AFL is V_MD and the one with an automatically-generated one is V_AD - a detailed description of the manual generation is described in Appendix in Section IX-A on page 15. Consider the cummulative coverage of V_AD vs. V_MD shown in Fig. 9 on page 16 in Appendix. For two programs (readelf and objdump), V_AD visits around 10% new lines and misses almost none of the lines visited by V_MD. For jhead, V_AD visits 130% new lines and misses none. For prettyprint, V_AD visits around 10% new lines but misses also around 10% of the lines visited by V_MD. For the remaining three programs, both fuzzers visit roughly the same lines. The automated dictionary performs well overall, and has the advantage of not requiring human knowledge. Note how for ar and unrtf we could not manually generate a dictionary (N/A in Fig. 9 on page 16), but the automatically-generated one (V_AD) improves over vanilla AFL (V_O3) by 27% and 17% respectively (Fig. 12f on page 18 and Fig. 12h on page 19).

In terms of bugs, V_AD and V_MD find 4 common bugs (bugs $3-6$ in Fig. 14 on page 20). V_AD finds 16 unique bugs not found by V_MD (bugs $7-22$); while V_MD finds 2 bugs not found by V_AD (bugs $1-2$). Of the 4 common bugs, 2 (bugs $5-6$) are found more consistently by V_AD and 2 (bugs $3-4$) more consistently by V_MD. In terms of total number of bugs found, V_AD is a stronger fuzzer. However, because V_MD finds two bugs that V_AD misses, running both fuzzers in parallel would be preferrable.

RQ4: How does an optimized dictionary (⋆_OD) perform vs. a non-optimized one (⋆_AD)?

To answer this question, we look at Fig. 10 on page 16 in Appendix. For three programs (readelf, objdump and pdftohtml), C_FBSP_OD visits between 4% and 20% new lines whilst missing a negligible number of lines C_FBSP_AD visits. On prettyprint, C_FBSP_OD visits 4% new lines but misses around 8% of lines visited by C_FBSP_AD. On



(a) Line coverage consistency graph for QSYM.

(b) Line coverage consistency graph for C_OD_FBSP.

Fig. 4: Line coverage consistency graphs for xmllint over 10 runs of 24 hours each.

three programs, C_FBSP_OD is worse than C_FBSP_AD and misses between 2% and 5% of lines C_FBSP_AD visits.

For bugs, C_FBSP_OD finds 8 bugs not found by C_-FBSP_AD; and C_FBSP_AD finds 1 bug not found by C_-FBSP_OD. C_FBSP_OD and C_FBSP_AD find 19 common bugs (Fig. 15 on page 20).

RQ5: How does controlled compilation affect custom passes?
To answer this question, we look at Fig. 11 on page 17 in Appendix. For five programs, there is no noticeable difference between LAF-INTEL and C_LAF-INTEL (LAF-INTEL with controlled compilation). For three programs, C_LAF-INTEL visits between 2% and 8% new lines whilst missing few lines visited by LAF-INTEL. For one program (xmllint), C_LAF-INTEL misses over 10% of lines LAF-INTEL visits whilst visiting just 2% new lines. Based on these results, we see that controlled compilation may help visit new parts of the code, but at the expense of missing some other parts that may be reachable without controlled compilation. This highlights the trade-off between execution speed and better coverage-guided feedback. This trend is confirmed by the bug discovery analysis (Fig. 16 on page 20): 12 bugs are common, 3 are only found by C_LAF-INTEL and 2 only found by LAF-INTEL.

RQ6: Does simply disabling compiler transformations boost fuzzing?
To answer this question, we look at the configuration V_O0 vs. the baseline vanilla AFL (V_O3) as shown in Fig. 8 on page 16 in Appendix. For three programs, V_O0 is worse than V_O3 as it misses many lines visited by V_O3 and covers no new lines (the histograms are only below zero). For three programs, V_O0 visits roughly the same number of lines but different parts of the program, i.e. both fuzzers complement each other. This illustrates the trade-off between program execution speed (V_O3) vs. better coverage-guided feedback (V_O0). The bug discovery analysis confirms this trend: 5 bugs are commonly found by each fuzzer, and they each find 1 bug the other fuzzer misses (Fig. 18 on page 21).

RQ7: Is there one best fuzzer for the job?
Based on coverage and bugs found, no single fuzzer emerges as the winner: A combination of fuzzers seems to be more effective. For example for readelf, Angora and C_OD together find all 8 distinct bugs but only 1 bug (bug 5) is common between them (Fig. 17 on page 21). If we had to select a single configuration though, C_FBSP_OD would be the best in terms of both coverage and bugs given our experimental results. It finds the largest number of bugs and finds most of the bugs for multiple programs (TABLE VI on page 17 in Appendix).

RQ8: What is a good fuzzing strategy?
We have already established that using a combination of complementary fuzzers/techniques is better than sticking with one single fuzzer. Another useful insight is the consistency with which lines are visited and bugs are found. Our results show that fuzzers have varying levels of consistency for different programs. These results can be used as guidelines to improve the fuzzing strategy: if a fuzzer is inconsistent, we may benefit from killing/restarting it every now and then. Furthermore, if we have two fuzzers with the same cumulative coverage, it is better to use the most consistent one.

RQ9: Is qualitative coverage a better measure than raw coverage numbers?
Based on the insights we have gained about fuzzers in previous sections, qualitative coverage provides advantages over a mere raw line number. The superiority of qualitative coverage can be illustrated with prettyprint (Fig. 12g on page 19). The configuration C_FBSP_OD visits about 13% new lines compared to the baseline V_O3, but it also fails to visit about 8% of lines that V_O3 visits. Had we used a quantitative coverage measure such as the total number of lines visited, we

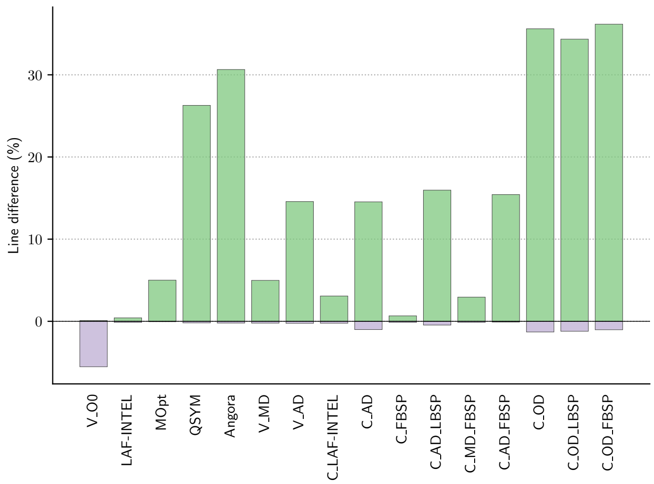


Fig. 5: Cumulative line coverage for `objdump -d @@` over 10 runs of 24 hours each. The baseline is `V_O3`.

would have concluded that `C_OD_FBSP` visits $13-8=5\%$ more lines that vanilla `V_O3`. This is incorrect and misleading. `C_OD_FBSP` is not better than `V_O3`. The two fuzzers are complimentary, as they visit different parts of the code. This shows that a qualitative measure is useful in practice to compare fuzzers.

D. Results on LAVA-M dataset

On the LAVA-M dataset [32], `C_FBSP_OD` outperforms recent fuzzers (except Redqueen) by finding more bugs (TABLE II). We were surprised by these results, which suggest most of the bugs in LAVA-M simply need guessing magic values copied directly from the input buffer (our automated dictionary manages to find most bugs). In general, we believe there are inherent limitations to using artificially-injected bug datasets for evaluating fuzzing.

Injecting bugs into a codebase inherently requires selecting *some* nodes in the CFG and injecting bugs in these. During evaluation, finding a bug then boils down to visiting this subset of the CFG nodes where bugs are injected. But how do we decide in which nodes to inject bugs? How do we decide if the selected nodes are representative of the overall program? This is a hard question. For example, in its current implementation, LAVA-M arbitrarily selects CFG nodes such that they are easy to reason about, in the sense that with high confidence the injected bugs do not alter the overall functionality of the program. Any bug-injecting tool must try to satisfy this constraint. Unfortunately this is hard and it limits the number of nodes where bugs may be injected in practice. This means that an evaluation based on a bug-injected dataset does not consider the program in its entirety, but biases the 'effectiveness' of a fuzzer to this small subset of arbitrarily selected CFG nodes.

On the contrary, coverage takes into account the entire program, not a limited, and arbitrary sample of the CFG node space. This is why we believe coverage may be a viable alternative to artificially-injected bugs. Arguments against coverage-based measures include the fact that a fuzzer may visit less code but find more bugs. This problem arises because the community has been thinking about coverage quantitatively rather than qualitatively. Using a qualitative measure, we can easily discern that two fuzzers visit different parts of the program, as we demonstrated in our evaluation.

Another argument against coverage-based metrics is that it is possible for a fuzzer to visit a basic block without triggering a bug present in the block. Again, this is not a deal breaker. It is straightforward to transform a bug-finding problem into a coverage-based problem, simply by using 'sanitizers', i.e. transformations to make the bug explicit in the control flow graph. For example, if we have a 10-byte long stack array, we can instrument the code to add a check to test the lookup index is within range, and crash the program if it is not. The instrumentation will create a new edge when the crash is triggered, hence it will become visible in the coverage metric. Note that there already exist instrumentations to detect different categories of bugs (ASAN, UBSAN, etc.), and we may build sanitizers based on these.

	uniq	base64	md5sum	who
`VUzzer`	27	17	N/A	50
`MOpt`[1]	27	39	23	5
`QSYM`	28	44	57	58
`T-Fuzz`	26	43	49	63
`Steelix`	7	43	38	194
`Angora`	28	44	57	1443
`NEUZZ`	29	48	60	1582
C_OD_FBSP	**28**	**44**	**49**	**1900**[2]
`Redqueen`	28	44	57	2134

[1] Paired with AFL.
[2] After 24 hours, `C_OD_FBSP` finds 2275 bugs in `who`.

TABLE II: Number of bugs found on LAVA-M dataset [32] after 5 hours (taken from corresponding papers).

VI. Lessons learned and guidelines

During our evaluation, we learned the following lessons which should be useful to other research teams:

1) Heuristics evaluation: Many papers we have surveyed use several heuristics to build a new fuzzer, but do not evaluate each heuristic in isolation. This makes it hard to understand which heuristic is most useful. In this work, we evaluate each transformation in isolation (e.g. dictionary, `FBSP`), before using them together in our final fuzzer.
2) AFL configuration: We should run AFL in a 'good' configuration, for example with a dictionary (manually and/or automatically generated by our toolchain) and in non-deterministic mode (`-d` option). Note that AFL's default (deterministic) mode is usually less effective that the non-deterministic mode (`-d` option) before the 24-hour mark [26], so it is important to run AFL for at least 24 hours. Also note that if the `-d` option is used, the dictionary will be ignored by AFL. So to test in non-deterministic mode with a dictionary, one must use parallel runs (`-M -x dict_file` and `-S`) or patch AFL itself

to run the dictionary even in non-deterministic mode (we implemented this when AFL is run with an optimized dictionary for our tests).

3) Coverage and bug consistency. This is an important measure to understand a fuzzer's behavior and adapt the fuzzing strategy. For example if a fuzzer is unstable, we may benefit from killing/restarting a fuzzing instance. Overall, the results indicate that fuzzers are complimentary to each other. This begs the question: which is the best combination for a given fuzzing budget?
4) Resource utilization is not well reported. Following previous point 3, ultimately we want to give a security researcher the possibility to select the best combination of tools/-fuzzers to run with a dedicated budget, e.g. based on CPU, RAM and duration (e.g. a few days before release or a few minutes for a commit). This is something we realized after we had finished our evaluation so we could not evaluate it. But we believe this is an important measure that must be added to fuzzing evaluations. For example, symbolic execution is more CPU-intensive and memory-hungry that fuzzing. Comparing three AFL instances vs. two AFL + one symbolic engine may not be a fair comparison. Symbolic execution may require as much CPU and memory as two AFL instances, or even more. Resource utilization is currently lacking in fuzzing evaluation papers (TABLE I on page 6), including ours. Running multiple instances of fuzzers to see how well they parallelize or complement each other is under-studied (TABLE I on page 6).
5) Ideally it would be useful to rank lines and bugs whether they are hard or easy to find. The research community may start tackling this problem empirically by testing several fuzzers and combination thereof. We may then give a difficulty score to lines and bugs based on the difficulty that existing techniques have to consistently find them. This would be a good way to gauge improvement over time, as we may update bug/line scores when new research is published.
6) Each research team currently runs their experiment on their own machines. Having a common infrastructure to run all fuzzers would help make results more comparable across research groups. It would also speed up advances in the field by not requiring each research team to re-benchmark all previously published tools.

VII. Related Work

There are three lines of research that relate to our work. One area is how compiler transformations affect security. Wang *et al.* [10] study how compilers' use of undefined behavior may remove security checks in code. Yang *et al.* [11] present a study of how cryptographic libraries try to evade compiler's dead-store-eliminations. Simon *et al.* [12] show that controlling side effects in cryptographic code cannot be guaranteed without compiler support. Another area close to our work is compiler techniques used to hinder fuzzing, which is studied by Jung *et al.* [33], Goransson [34] and Whitehouse [35]. This is essentially the opposite of what we do in this work. The third area close to our work is fuzzing in general. Model-based fuzzers such as Peach [36] try to build a model of a protocol/format. SmartSeed [4] and Skyfire [3] use data-driven techniques to learn a format model automatically. Some fuzzers generate input based on context-free grammar, such as Godefroid *et al.* [37], LangFuzz [38], lava [39], Ifuzzer [40] and CSmith [41]. Mutation-based fuzzers mutate their input without the help of a grammar or model. In this context, Rebert *et al.* [42], Woo *et al.* [43], FairFuzz [26], VUzzer [6], MOpt [27] SlowFuzz [44] and AFLFast [45] study the seed scheduling algorithms. To solve difficult path constraints, Angora [5] and NEUZZ [20] use ML techniques; T-Fuzz [7] removes conditions from the original binary; Driller [22], DART [46], SAGE[47], SymFuzz [48], Taintscope [49] and QSYM [17] use fuzzing in combination with symbolic execution. Redqueen [8]'s insight is that most programs use untransformed input data for path conditions, so it uses heuristics for this case in order to avoid costly symbolic execution. HonggFuzz [15] and kAFL [24] use modern CPU extensions to improve fuzzing. CollAFL [1] and Instrim [50] attempt to reduce the probability of edge collision for coverage-guided fuzzing. Xu *et al.* [51] devise OS primitives to scale fuzzing. Padhye *et al.* [52] devise a framework to build domain-specific fuzzing applications. Klees *et al.* [30] try to improve fuzzing evaluation and propose guidelines. Chen *et al.* [53] study how to combine fuzzers to improve their efficacy.

VIII. Conclusion

We observed that compiler transformations harm fuzzing because they alter the basic block layout a fuzzer depends on to make progress. So we proposed 'controlled compilation', a set of techniques to transform the source code and IR of a program to be more fuzzing-friendly. We showed through evaluation that controlled compilation helps a fuzzer to visit different parts of a program and uncover new bugs. Controlled compilation often outperforms recent fuzzing techniques in terms of program coverage and number of bugs found. A benefit of controlled compilation is that it does not require advanced fuzzing techniques and it may be used with an off-the-shelf fuzzer like AFL. This suggests that recent advances in fuzzing are more limited that previously thought.

We identified two main reasons to explain why. First, it has proven difficult for researchers to appropriately configure existing fuzzers such as AFL. To address this problem, we provided guidelines and new LLVM passes to help automate AFL's configuration. We hope this will help researchers to perform a fairer comparison with AFL.

Second, we found that coverage-based measures used in previous research lose valuable information such as *which* parts of a program a fuzzer actually visits and how *consistently* it does so. To address this, we proposed a rigorous evaluation methodology based on 'qualitative coverage'. Qualitative coverage uniquely identifies each program line in order to help understand which lines are commonly visited by different fuzzers vs. which lines are visited only by a particular fuzzer.

This evaluation methodology helps to understand if a fuzzer performs better, worse, or is complementary to another fuzzer. This may help security practitioners adjust their fuzzing strategies.

REFERENCES

[1] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 679–696, IEEE, 2018.

[2] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu, "Fot: a versatile, configurable, extensible fuzzing framework," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 867–870, ACM, 2018.

[3] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Security and Privacy (SP), 2017 IEEE Symposium on*, pp. 579–594, IEEE, 2017.

[4] C. Lv, S. Ji, Y. Li, J. Zhou, J. Chen, P. Zhou, and J. Chen, "Smartseed: Smart seed generation for efficient fuzzing," *arXiv preprint arXiv:1807.02606*, 2018.

[5] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 711–725, IEEE, 2018.

[6] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing.," in *NDSS*, vol. 17, pp. 1–14, 2017.

[7] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 697–710, IEEE, 2018.

[8] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence,"

[9] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," *arXiv preprint arXiv:1709.07101*, 2017.

[10] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined behavior: what happened to my code?," in *Proceedings of the Asia-Pacific Workshop on Systems*, p. 9, ACM, 2012.

[11] Z. Yang, B. Johannesmeyer, A. T. Olesen, S. Lerner, and K. Levchenko, "Dead store elimination (still) considered harmful," in *26th USENIX Security Symposium. USENIX Association*, 2017.

[12] L. Simon, D. Chisnall, and R. Anderson, "What you get is what you c: Controlling side effects in mainstream c compilers," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 1–15, IEEE, 2018.

[13] "laf-llvm-pass." https://gitlab.com/laf-intel/laf-llvm-pass.

[14] "LLVMs Analysis and Transform Passes." https://llvm.org/docs/Passes.html.

[15] "Hongfuzzer." https://github.com/google/honggfuzz.

[16] M. Zalewski, "American fuzzy lop (afl) fuzzer."

[17] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 745–761, 2018.

[18] "IfConversion pass." https://llvm.org/doxygen/IfConversion_8cpp_source.html.

[19] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 627–637, ACM, 2017.

[20] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program learning," *arXiv preprint arXiv:1807.05620*, 2018.

[21] U. Kargén and N. Shahmehri, "Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 782–792, ACM, 2015.

[22] 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.," in *NDSS*, vol. 16, pp. 1–16, 2016.

[23] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344, ACM, 2017.

[24] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for {OS} kernels," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 167–182, 2017.

[25] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2345–2358, ACM, 2017.

[26] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 475–485, ACM, 2018.

[27] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{MOPT}: Optimized mutation scheduling for fuzzers," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1949–1966, 2019.

[28] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[29] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, "V-fuzz: Vulnerability-oriented evolutionary fuzzing," *arXiv preprint arXiv:1901.01142*, 2019.

[30] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2123–2138, ACM, 2018.

[31] "When Results Are All That Matters: The Case of the Angora Fuzzer ." https://andreas-zeller.blogspot.com/2019/10/when-results-are-all-that-matters-case.html.

[32] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 110–121, IEEE, 2016.

[33] J. Jung, H. Hu, D. Solodukhin, K. H. Lee, and T. Kim, "Fuzzification: Anti-fuzzing techniques," in *27th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019.

[34] D. GÖRANSSON and E. EDHOLM, "Escaping the fuzz," 2016.

[35] O. Whitehouse, "Introduction to Anti-Fuzzing: A Defence in Depth Aid." https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2014/january/introduction-to-anti-fuzzing-a-defence-in-depth-aid/.

[36] M. Eddington, "Peach fuzzing platform," *Peach Fuzzer*, vol. 34, 2011.

[37] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM Sigplan Notices*, vol. 43, pp. 206–215, ACM, 2008.

[38] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pp. 445–458, 2012.

[39] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," in *ACM SIGPLAN Notices*, vol. 35, pp. 1–13, ACM, 1999.

[40] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *European Symposium on Research in Computer Security*, pp. 581–601, Springer, 2016.

[41] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *ACM SIGPLAN Notices*, vol. 46, pp. 283–294, ACM, 2011.

[42] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 861–875, 2014.

[43] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 511–522, ACM, 2013.

[44] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2155–2168, ACM, 2017.

[45] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, 2017.

[46] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, pp. 213–223, ACM, 2005.

[47] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

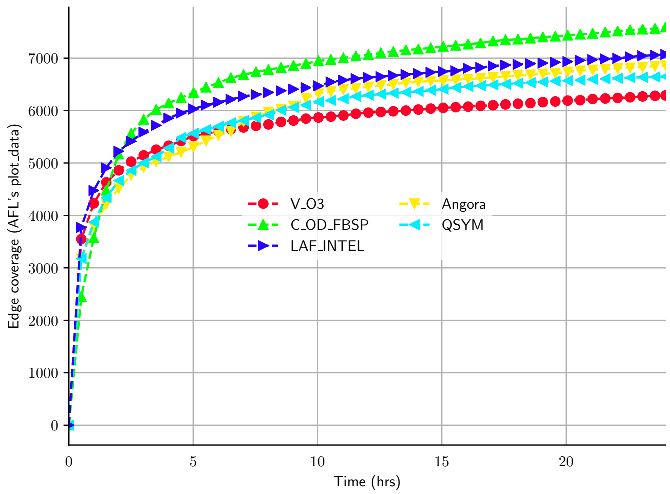


Fig. 6: Coverage trend (total number of edges) over 24 hours for `objdump -d @@`

Name	Description
`V_O3`	three **V**anilla AFL
`V_O0`	three **V**anilla AFL without compiler optimizations
`LAF_INTEL`	two **V**anilla AFL + one `LAF_INTEL` build
`MOpt`	three MOpt AFL [27]
`QSYM`	two vanilla AFL + one `QSYM`[1]
`Angora`	two vanilla AFL + one `Angora`[1]
`V_MD`	three **V**anilla AFL with **M**anual **D**ictionary
`V_AD`	three **V**anilla AFL with **A**utomated **D**ictionary
`C_LAF_INTEL`	two **V**anilla AFL + one `LAF_INTEL` using `CC`
`C_AD`	two **V**anilla AFL + one `CC` build using **A**utomated **D**ictionary
`C_FBSP`	one vanilla AFL + two **C**ompanions: - one `CC` build - one **F**ull **B**yte **S**plitting **P**ass build
`C_AD_LBSP`	one vanilla AFL + two **C**ompanions: - one `CC` build - one **L**ight **B**yte **S**plitting **P**ass build All instances use an **A**utomated **D**ictionary
`C_MD_FBSP`	one vanilla AFL + two **C**ompanions: - one `CC` build - one **F**ull **B**yte **S**plitting **P**ass build All instances use an **M**anual **D**ictionary
`C_AD_FBSP`	one vanilla AFL + two **C**ompanions: - one `CC` build - one **F**ull **B**yte **S**plitting **P**ass build All instances use an **A**utomated **D**ictionary
`C_OD`	one vanilla AFL + two `CC` build **C**ompanions All instances use an **O**ptimized **D**ictionary
`C_OD_LBSP`	one vanilla AFL + two **C**ompanions: - one `CC` build - one **L**ight **B**yte **S**plitting **P**ass build all instances use an **O**ptimized **D**ictionary
`C_OD_FBSP`	one vanilla AFL + two **C**ompanions: - one `CC` build - one **F**ull **B**yte **S**plitting **P**ass build all instances use an **O**ptimized **D**ictionary

[1] this is the configuration suggested by the authors of the tool

TABLE III: Programs used for fuzzing. `CC` is an abbreviation for Controlled Compilation.

[48] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*, pp. 725–741, IEEE, 2015.

[49] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*, pp. 497–512, IEEE, 2010.

[50] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang, "Instrim: Lightweight instrumentation for coverage-guided fuzzing," in *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[51] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2313–2328, ACM, 2017.

[52] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "Fuzzfactory: domain-specific fuzzing with waypoints," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, p. 174, 2019.

[53] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1967–1983, 2019.

[54] "Executable and Linkable Format (ELF)." http://www.skyfree.org/linux/references/ELF_Format.pdf.

[55] "Token capture via an llvm-based analysis pass." https://doar-e.github.io/blog/2016/11/27/clang-and-passes/.

IX. Appendix

A. Manual Dictionary Generation

The dictionaries were taken from AFL's `dict` folder ('AFL' in column 'dict' of TABLE IV on the next page). For certain programs such as `ar`, it was too time-consuming to manually create a dictionary and AFL does not provide one. So we could not evaluate fuzzing with a manual dictionary ('N/A' in column 'dict'). For ELF-parsing binaries `objdump` and `readelf`, AFL does not provide a dictionary but we were able to manually create one in a few hours. For this, we *1)* manually extracted constant strings from the ELF standard [54], *2)* used the results of the command `which ls | grep '\.'` and *3)* used an LLVM pass from an online blog [55]. We concatenated all the magic values returned by the three steps to generate the dictionary: we call this the 'bundle' generation in column 'dict' of TABLE IV on the following page.

Executable	Version	Project	Arguments	Dict
prettyprint	0.10.1	gumbo-html	@@	AFL
ar	2.27	binutils	x @@	N/A
djpeg	6b	libjpeg	@@	AFL
pdftohtml	4.00	xpdf	@@	AFL
xmllint	2.9.8	libxml2	@@	AFL
jhead	3.02	jhead	@@	AFL
libpng_read_fuzzer	1.6.35	libpng	@@	AFL
readelf	2.27	binutils	-a @@	bundle
objdump	2.27	binutils	-d @@	bundle
cxxfilt	2.27	binutils		N/A
unrtf	0.20.4	unrtf	@@	N/A

TABLE IV: Programs used for fuzzing and the dictionary used. 'AFL' means the dictionary is the one shipped with AFL. 'bundle' is described in Section IX-A on the preceding page.

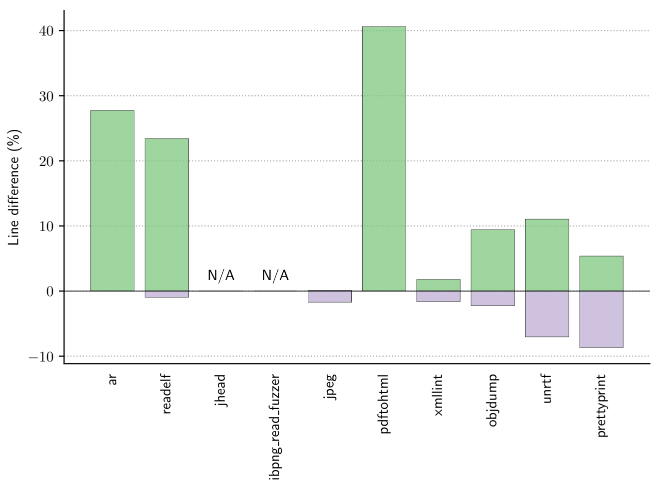


Fig. 7: Comparison of cumulative line coverage of C_OD_-FBSP vs. baseline QSYM.

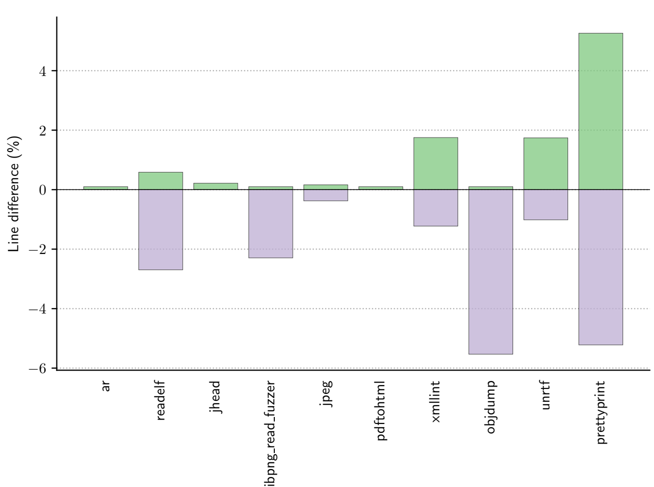


Fig. 8: Comparison of cumulative line coverage of V_O0 vs. baseline V_O3.

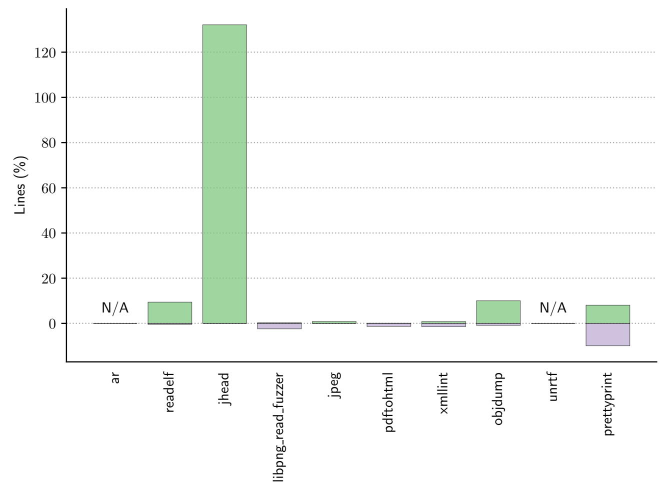


Fig. 9: Comparison of cumulative line coverage of V_AD vs. baseline V_MD.

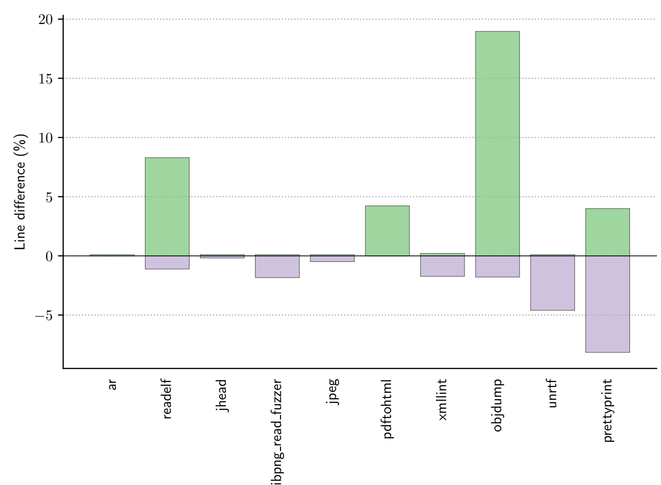


Fig. 10: Comparison of cumulative line coverage of C_-FBSP_OD vs. baseline C_FBSP_AD.

	Total bugs found	AFL (V_O3)	V_O0	LAF-INTEL	Mopt	QSYM	Angora	V_MD	V_AD	C_LAF-INTEL	C_FBSP	C_AD_LBSP	C_MD_FBSP	C_AD_FBSP	C_OD	C_OD_LBSP	C_OD_FBSP
ar	4	0	0	0	0	0	1	NA	2	0	2	1	NA	2	1	1	1
readelf	8	0	0	0	0	0	4	4	4	0	0	2	5	3	5	5	6
objdump	4	0	0	0	0	3	1	0	0	0	0	0	0	0	2	2	4
pdftohtml	2	1	1	1	NA	1	1	1	1	1	1	1	1	1	1	1	2
unrtf	7	4	5	6	4	4	4	NA	5	5	4	6	NA	6	4	4	6
jhead	11	1	0	7	0	5	8	1	8	9	6	8	7	8	7	8	8
total	36	6	6	14	4	13	19	6	20	15	13	18	13	20	20	21	27

TABLE V: Number of bugs found

	Total bugs found	AFL (V_O3)	V_O0	LAF-INTEL	Mopt	QSYM	Angora	V_MD	V_AD	C_LAF-INTEL	C_FBSP	C_AD_LBSP	C_MD_FBSP	C_AD_FBSP	C_OD	C_OD_LBSP	C_OD_FBSP
Illegal Memory Access (read)	18	1	1	5	1	5	8	1	6	7	4	4	5	4	7	8	12
Buffer Overflow (write)	4	1	0	2	0	2	2	1	2	0	3	3	2	3	2	2	3
Unsafe Signed to Unsigned Conversion	1	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1
Unsafe long to int conversion	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1
Integer Overflow	3	0	0	2	0	1	2	0	2	3	1	2	2	2	2	2	2
Double Free	2	0	0	0	0	0	1	0	2	0	0	1	0	2	1	1	1
Null Pointer Dereference	6	2	3	2	2	2	3	3	5	2	2	5	3	6	5	5	6
Uncaught Exceptions	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
total	36	6	6	14	4	13	19	6	20	15	13	18	13	20	20	21	27

TABLE VI: Classification of bugs discovered by fuzzers.

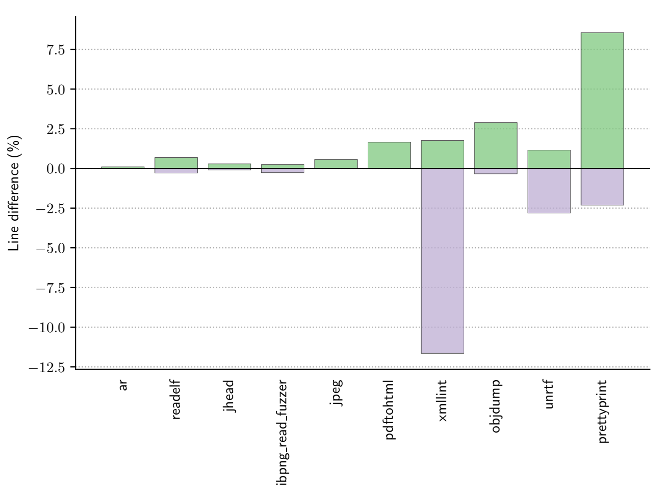
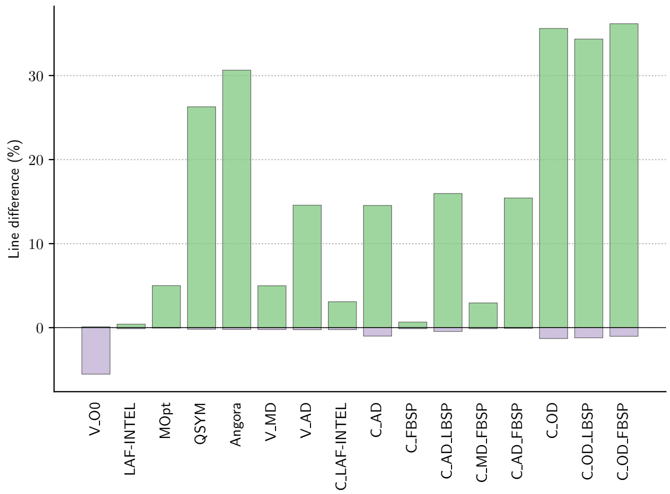
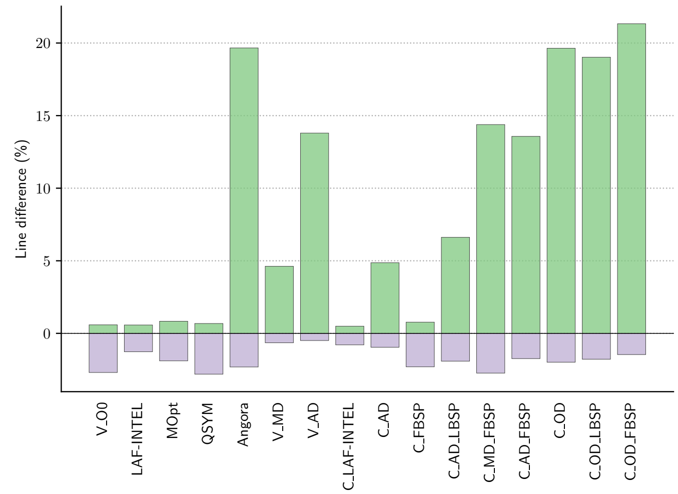


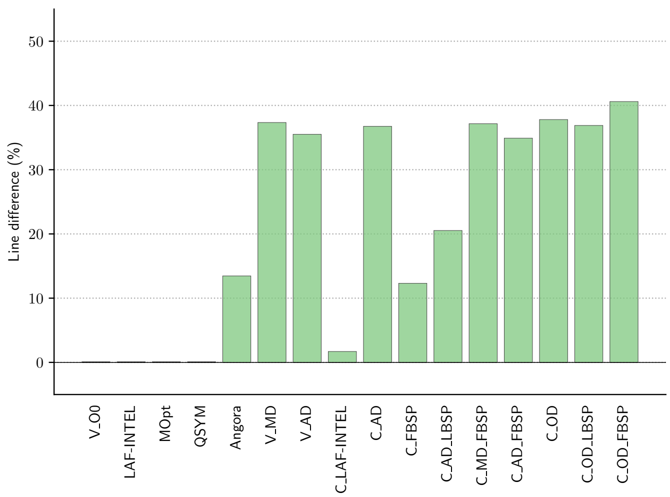
Fig. 11: Comparison of cumulative line coverage of C_LAF-INTEL vs. baseline LAF-INTEL.



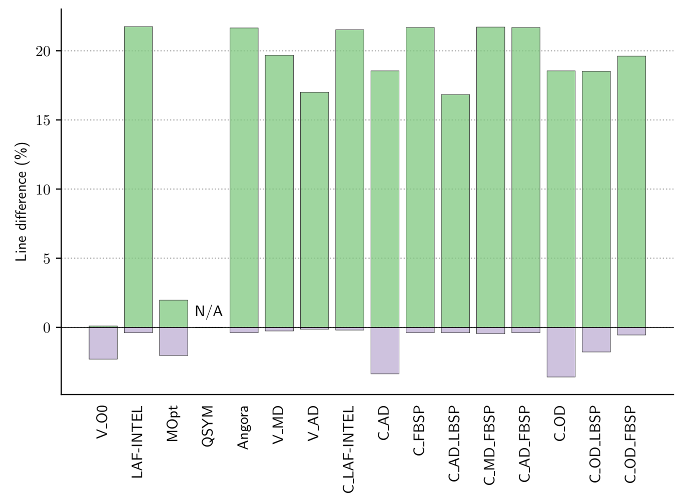
(a) `objdump -d @@`



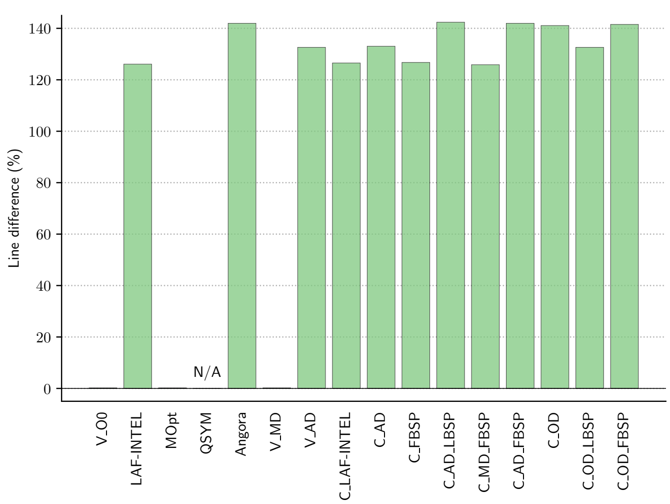
(b) `readelf -a @@`



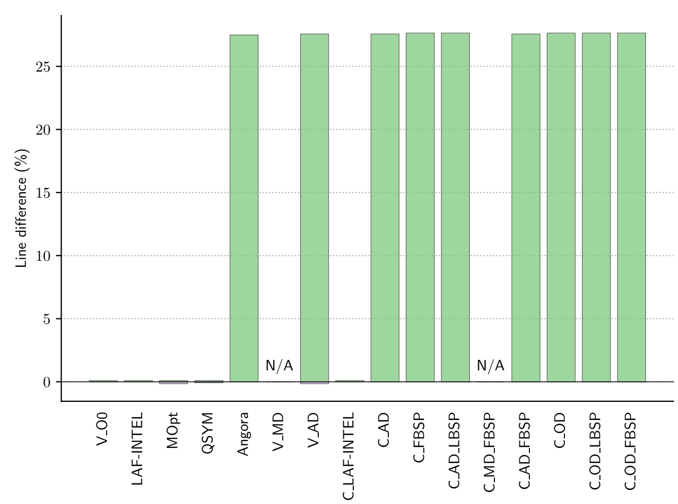
(c) `pdftohtml @@`



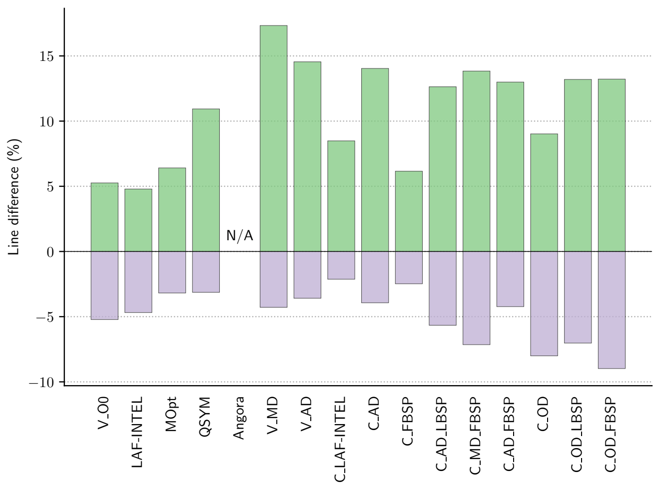
(d) `libpng_read_fuzzer @@`



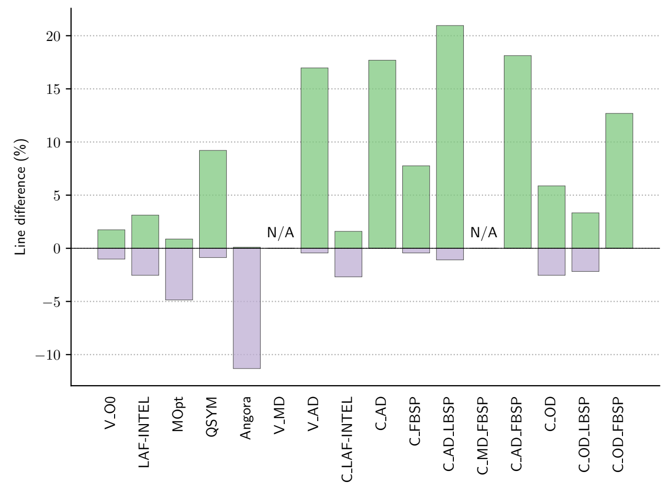
(e) `jhead @@`



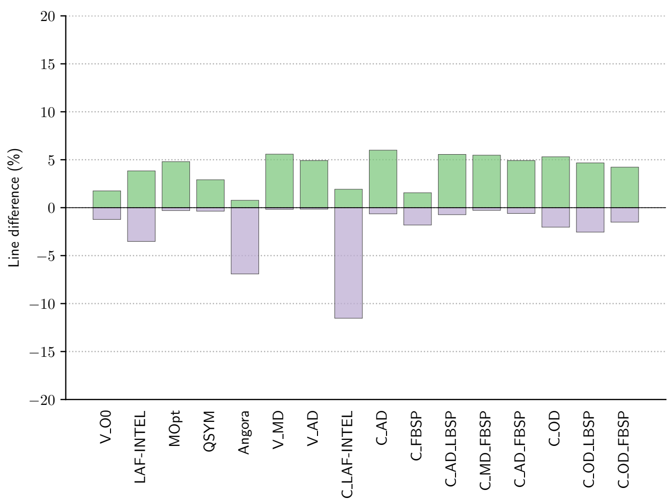
(f) `ar x @@`



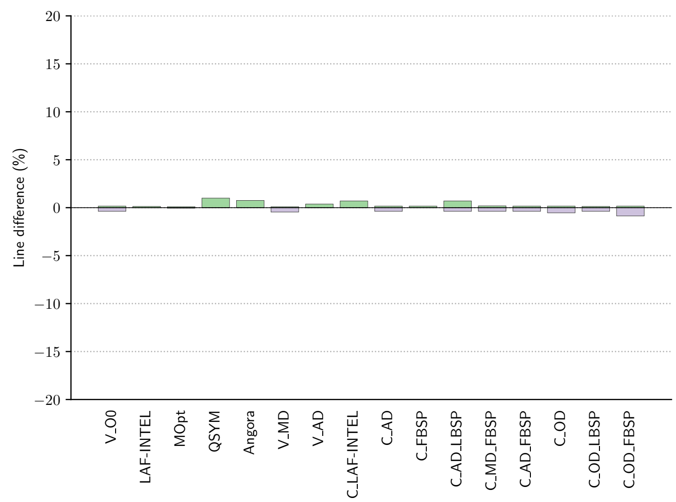
(g) `prettyprint @@`



(h) `unrtf @@`



(i) `xmllint @@`



(j) `djpeg @@`

Fig. 12: Cumulative line coverage over 10 runs of 24 hours each. The baseline is `V_O3`.

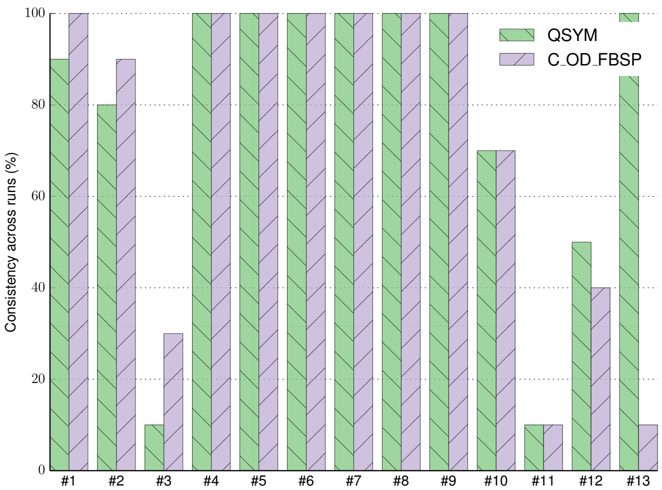


Fig. 13: Comparison of bug discovery consistency of QSYM vs C_FBSP_OD. Only common bugs are shown.

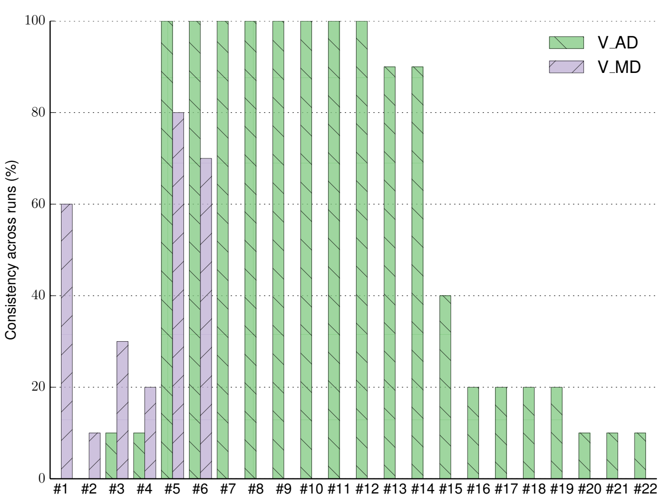


Fig. 14: Comparison of bug discovery consistency of V_AD vs V_MD. All bugs are shown.

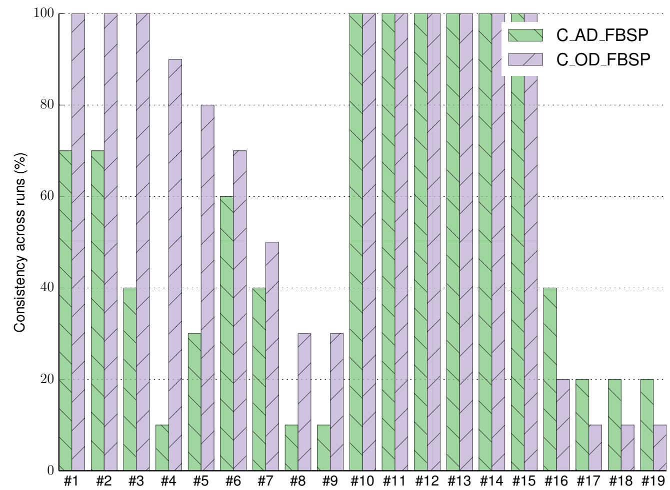


Fig. 15: Comparison of bug discovery consistency of C_-FBSP_AD vs. C_FBSP_OD. Only common bugs are shown.

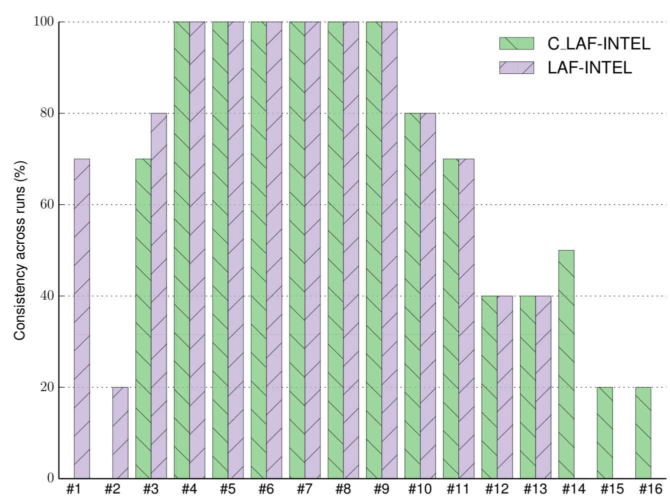


Fig. 16: Comparison of bug discovery consistency of LAF-INTEL vs C_LAF-INTEL. All bugs are shown.

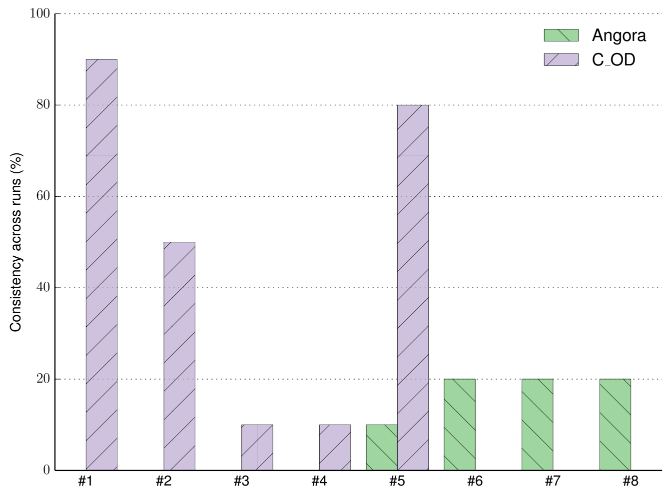


Fig. 17: Comparison of bug discovery consistency for `readelf` of `Angora` vs. `C_OD`. All bugs are shown.

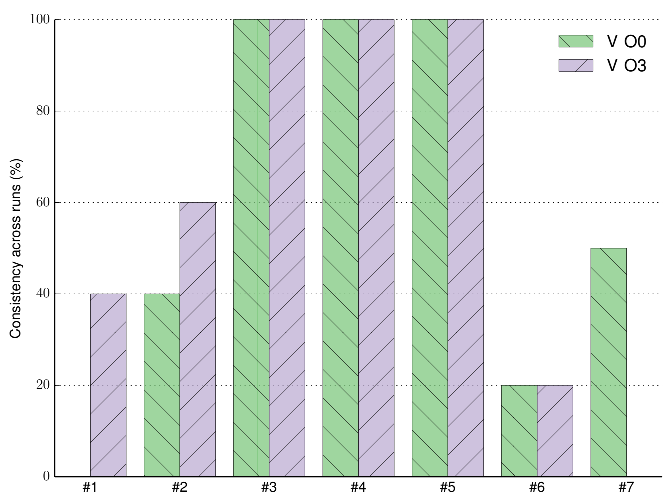


Fig. 18: Comparison of bug discovery consistency of `V_O0` vs. `V_O3`. Only common bugs are shown.